Documentacion

Modulos (overview):

-main : Arranca el ejecutable, toma argumento -d. Abre el .vmx, inicializa vm y llama a vm.run. No ejecuta instrucciones, solo “prepara en entorno y dispara la ejecucion”.

-vm: Implementa el ciclo fetch->decode->execute. Inicializa registros/segmentos y lleva el IP. Pide al decoder interpretar la instruccion y al cpu ejecutarla.

- decoder: Interpreta las instrucciones. Lee bytes del segmento de codigo a partir de IP, extrae opcode, tipos y operandos.

-disasm: Convierte la instruccion decodificada en texto legible. Traduce las instrucciones a codigo assembler

-memory: Modela la RAM y los segmentos. Valida rangos para evitar accesos fuera de segmento y provee helpers de acceso.

-cpu: Implementa las instrucciones MOV/ADD/SUB/MUL/DIV/CMP, AND/OR/XOR/NOT, SHL/SHR/SAR, saltos (JMP/JZ/JNZ/JN/JNN/JP/JNP), LDL/LDH, RND, SWAP y SYS 1/2. Lee/escribe operandos.Actualiza los flags N y Z.

Funciones (por modulo):

main:

-int main(int argc, char** argv): Valida argumentos, reconoce el flag “-d”. Crea e inicializa una instancia de VM(vm_init), carga el binario .vmx en memoria(vm_load) y llama a vm_run.

vm:

-static inline u16 read_u16_be(const u8 b[2]): Lee un entero de 16 bits en big-endian desde dos bytes.

-void vm_init(VM* vm, bool disassemble): inicializa la vm y deja configurado el modo de desensamblado.

-bool vm_load(VM* vm, const char* path): Carga un programa .vmx desde disco y configura segmentos/registros base. Abre el archivo, valida nombre y version. Lee el code_size y verifica que no supere el ram_size ni sea nulo. Copia el bloque de codigo, configura cementos y fija los registros logicos base (CS,DS,IP)

-int vm_run(VM* vm): Ejecuta el programa cargado realizando fetch -> decode ->(opcional) disasm-> execute en bucle. No implementa la lógica de las instrucciones; sólo coordina y controla errores.

memory:

-static inline void set_lar_mar(VM* vm, u16 seg_idx, u16 offset, u16 nbytes, u16 phys): Actualiza LAR y MAR, no toca RAM.

-bool translate_and_check(VM* vm, u16 seg_idx, u16 offset, u16 nbytes, u16* out_phys): traduce a direccion fisica y valida que el rango entre completo dentro del segmento.

-static bool read_bytes(VM* vm, u16 seg_idx, u16 offset, void* dst, u16 nbytes): Lee nbytes del segmento dado (DATOS) a un buffer, actualizando LAR/MAR/MBR.

-static bool write_bytes(VM* vm, u16 seg_idx, u16 offset, const void* src, u16 nbytes): Escribe nbytes desde un buffer al segmento (DATOS), actualizando MAR/MBR.

-mem_read_u8/mem_read_u16/mem_read_u32: Lee 1/2/4 bytes (datos) en formato big-endian y los retorna en out.

-mem_write_u8/mem_write_u16/mem_write_u32:Escribe 1/2/4 bytes (datos) en big-endian

-bool code_read_bytes(VM* vm, u16 phys,void* dst, u16 nbytes): Lee bytes desde el segmento de codigo por dirección física.

cpu:

-static inline uint16_t hi16_u32(uint32_t x)/t16_t lo16_u32(uint32_t x): toma los 16 bits mas altos/bajos de un uint 32.

-static inline uint32_t shamt32(uint32_t v): (shift amount 32) se queda con los 5 bits bajos del valor para que el shift siempre esté en el rango [0..31] y evitar el comportamiento indefinido de C

-static inline uint32_t cc_Nbit(VM* vm)/static inline uint32_t cc_Zbit(VM* vm): Leen los flags N y Z desde CC

-static inline uint16_t ecx_count(uint32_t ecx)/static inline uint16_t ecx_size (uint32_t ecx): Separa ECX para obtener el tamaño de celda y la cantidad de celdas para sys ½.

-static inline void jump_to_code(VM* vm, uint16_t off): helper para implementar saltos dentro del codigo.

-static inline uint8_t vm_regidx_from_vmxcode(uint8_t code): mapea el codigo del registro del .vmx a mi enumerador interno (vm.h).

-static bool get_mem_address(VM* vm, const DecodedOp* op, u16* seg_idx, u16* offset): Calcula seg:off tomando el operando decodificado de tipo OT_MEM

-bool read_operand_u32(VM* vm, const DecodedOp* op, uint32_t* out):Lee un operando (32 bits lógicos) cualquiera.

-bool write_operand_u32(VM* vm, const DecodedOp* op, uint32_t val):Escribe un valor (32 bits) en REG o MEM.

-static bool phys_of_cell(VM* vm, u32 base_ptr, u16 cell_size, u16 idx, u16* out_phys):Traduce la celda #idx desde un puntero base lógico (EDX) y tamaño (1/2/4).

-static bool mem_read_cell(VM* vm, u16 seg, u16 offset, u16 n, u32* out):Lectura de celda (1/2/4) usando las funciones de memoria.

-static bool mem_write_cell(VM* vm, u16 seg, u16 offset, u16 n, u32 val):Escritura de celda (1/2/4) usando las funciones de memoria.

-static bool read_jump_offset(VM* vm, const DecodedInst* di, int16_t* off):toma el operando A de un salto, obtiene los 16 bits de destino absoluto en CS

-static bool read_line(char* buf, size_t cap): Lee una linea desde stdin al buffer, y la deja sin \n ni \r al final.

-static bool parse_input(u32 mode, u16 cell_size, u32* out): Convierte el texto leído con read_line a un valor de 32 bits según el modo y el tamaño de celda.

-static void print_binary(u32 v): Imprime el valor en binario con prefijo 0b y sin ceros a la izquierda.

-static void print_chars(u32 v, u16 size): Imprime “size” caracteres contenidos en “v”, interpretando el entero en big-endian.

-static void print_cell(u32 modes,u32 value, u16 cell_size): Imprime una celda según la máscara de modos (BIN/HEX/OCT/DEC/CHR)

-void init_dispatch_table(OpHandler tb[256]):Inicializa la tabla de handlers por opcode.

-int exec_instruction(VM* vm, const DecodedInst* di, OpHandler tb[256]): Ejecuta una instrucción llamando el handler correspondiente a di->opcode.