SIMPLEZ-13

Virus Payal

Emulador de CPU Simplez-13 — Documento Técnico Ampliado

1-Introducción General

¿Qué es Simplez?

Simplez-13 es una **CPU didáctica** diseñada para enseñar los fundamentos de la arquitectura de computadores. No existe físicamente; es un modelo conceptual que simplifica al máximo las operaciones reales de un procesador. El objetivo de este emulador es **replicar el comportamiento del hardware** utilizando **lenguaje C**.

Estructura básica de la CPU Simplez-13

Componente	Función principal	Representación en el emulador
Memoria	Guarda instrucciones y datos	`uint16_t mem[4096]`
AC (Acumulador)	Realiza operaciones aritméticas	`uint16_t acc`
X (Registro índice)	Permite direccionamiento avanzado	`uint16_t x`
PC (Program Counter)	Indica la siguiente instrucción	`uint16_t pc`
UC (Unidad de Control)	Coordina el ciclo de ejecución	`loop()` en main.c
ALU (Unidad Aritmético-Lógica)	Realiza sumas, restas, etc.	Funciones en instrucciones.c

Características clave de Simplez-13

- Memoria: 4096 palabras de 12 bits.
- Registros: AC, X y PC.
- Juego de instrucciones mejorado con operandos inmediatos (LD #, SUB #).
- Ciclo de control basado en **Fetch → Decode → Execute**.

2-Modelo Estructural (Hardware Virtual)

2.1 Memoria Principal (RAM virtual)

La memoria principal almacena tanto **datos** como **instrucciones**. En Simplez-13, cada celda de memoria contiene una palabra de **12 bits**, pero en C se representa con un entero de 16 bits (uint16_t) por comodidad.

```
#define MEM_SIZE 4096
uint16_t mem[MEM_SIZE];
```

Cada instrucción o dato se guarda como un número entero. Por ejemplo:

```
mem[0] = 0xA87; // Instrucción codificada en binario
mem[1] = 0x003; // Dato literal o dirección
```

La memoria se trata como un **vector indexado**, y el **PC** (contador de programa) apunta siempre a la siguiente posición a ejecutar.

2.2 Registros (Variables Globales de la CPU)

```
uint16_t acc = 0;  // Acumulador
uint16_t x   = 0;  // Registro índice
uint16_t pc  = 0;  // Contador de programa
```

- **AC** (acumulador): donde se realizan las operaciones matemáticas y lógicas.
- **X**: usado para **direccionamiento indexado**.
- **PC**: contiene la dirección actual de ejecución; se incrementa automáticamente tras cada instrucción (salvo saltos o interrupciones).

2.3 Registro de Estado (Flags)

El registro de estado mantiene **condiciones del procesador** tras cada operación.

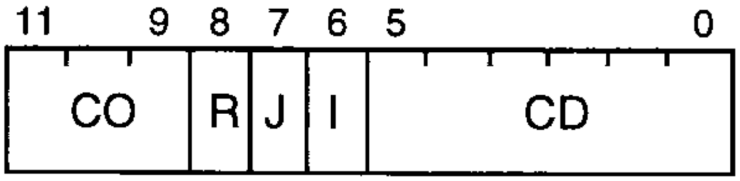
```
struct status {
    uint8_t z : 1; // Flag Zero
    uint8_t h : 1; // Flag Halt
} status;
```

- **Z (Zero):** se activa si el resultado de la última operación fue 0. Se usa en instrucciones como BZ (Branch if Zero).
- **H (Halt):** indica que el emulador debe detener la ejecución.

3- Modelo Funcional (Formato de Instrucción y Decodificación)

Cada instrucción en Simplez-13 tiene **12 bits** que se dividen en campos:

Bits	Campo	Descripción
11–9	CO	Código de operación (opcode)
8	R	Registro (0 = AC, 1 = X)
7–6	J/I	Modo de direccionamiento
5–0	CD	Campo de datos o dirección



Código de Operación (binario)	Instrucción	Tipo de Operación	Bit 8 (R)	Modo de Direccionamiento	Referencia/Comentario
000	ST (Store)	Operación de memoria	Significativo	Determinado por la tabla del apartado 2	Usa modos Directo, Indirecto, Indexado, etc.
001	LD (Load)	Operación de memoria	Significativo	Determinado por la tabla del apartado 2	Igual que ST, lee datos desde memoria.
010	ADD (Suma)	Operación aritmética	Significativo	Determinado por la tabla del apartado 2	Usa la EA para obtener el operando a sumar.
011	BR (Branch)	Instrucción de salto	Indiferente	Determinado por la tabla del apartado 2	El bit 8 no afecta al salto.

100	BZ (Branch if Zero)	Instrucción condicional	Indiferente	Determinado por la tabla del apartado 2	Igual que BR, pero condicionado al flag Z.
101	LD # (Carga literal)	Operación inmediata	Significativo	**Inmediato (CD = valor literal)**	Ignora los modos del apartado 2.
110	SUB # (Resta literal)	Operación inmediata	Significativo	**Inmediato (CD = valor literal)**	El CD contiene directamente el valor a restar.
111	HALT (Detener CPU)	Control	—	—	Detiene el ciclo de ejecución.

Ejemplo:

0xA87 → 1010 1000 0111 → LD # AC, 7

Decodificación en C:

```c
uint8_t op   = (inst >> 9) & 0x7;  // Opcode
uint8_t reg  = (inst >> 8) & 0x1;  // Registro destino
uint8_t dirm = (inst >> 6) & 0x3;  // Modo de direccionamiento
uint16_t cd  = inst & 0x3F;        // Campo de datos
```

Estos desplazamientos y máscaras (>>, &) extraen cada parte de la instrucción de forma eficiente.

4- Unidad de Control (Ciclo de Ejecución)

La **Unidad de Control (UC)** se encarga del ciclo clásico **Fetch → Decode → Execute**:

```c
void loop() {
    while (!status.h) { // Mientras no se haya detenido
        uint16_t inst = mem[pc]; // FETCH
        uint8_t op   = (inst >> 9) & 0x7;
        uint8_t reg  = (inst >> 8) & 0x1;
        uint8_t dirm = (inst >> 6) & 0x3;
        uint16_t cd  = inst & 0x3F;

        uint16_t data = direccionar(dirm, cd); // DECODE

        iset[op](reg, data); // EXECUTE
        pc++;
    }
}
```

Explicación paso a paso:

1. **FETCH:** se lee la instrucción de memoria apuntada por pc.
2. **DECODE:** se separa el binario en sus partes (opcode, modo, registro, datos).
3. **EXECUTE:** se llama a la función correspondiente en el vector iset[].
4. **UPDATE:** se incrementa pc (salvo que haya un salto o HALT).

5- Unidad Aritmético-Lógica (UAL o ALU)

La **ALU** contiene las funciones que ejecutan cada instrucción. Todas están en instrucciones.c, y se asocian con su opcode mediante un vector de punteros a funciones:

```
void (*iset[])(uint8_t reg, uint16_t data) = {
    st, ld, add, br, bz, ld_sharp, sub_sharp, halt
};
```

Cada índice del vector corresponde a un **opcode**.

5.1 Función LD (Load)

Carga un valor de memoria en un registro.

```
void ld(uint8_t reg, uint16_t ea) {
    if (reg == 0) acc = mem[ea];
    else x = mem[ea];
    status.z = (reg ? x : acc) == 0;
}
```

📌 *Efecto:* Lee desde memoria y actualiza el flag Z si el resultado es 0.

5.2 Función ST (Store)

Guarda el contenido de un registro en memoria.

```
void st(uint8_t reg, uint16_t ea) {
    mem[ea] = (reg ? x : acc);
}
```

📌 *Efecto:* Escribe el valor del registro seleccionado (AC o X) en la dirección ea.

5.3 Función ADD (Suma)

Suma el contenido de memoria al registro especificado.

```
void add(uint8_t reg, uint16_t ea) {
    if (reg == 0) acc += mem[ea];
    else x += mem[ea];
    status.z = ((reg ? x : acc) == 0);
}
```

📌 *Efecto:* Actualiza el acumulador o X y modifica el flag Z.

5.4 Función SUB # (Resta inmediata)

Resta un valor literal al registro.

```
void sub_sharp(uint8_t reg, uint16_t data) {
    if (reg == 0) acc -= data;
    else x -= data;
    status.z = ((reg ? x : acc) == 0);
}
```

📌 *Efecto:* Usa operandos inmediatos en lugar de acceder a memoria.

5.5 Función BZ (Branch if Zero)

Salta si el flag Z está activado.

```
void bz(uint8_t reg, uint16_t addr) {
    if (status.z) pc = addr - 1; // -1 para compensar el ++ del loop
}
```

📌 *Efecto:* Modifica el flujo de ejecución según el resultado anterior.

5.6 Función HALT

Detiene la CPU.

```
void halt(uint8_t reg, uint16_t data) {
    status.h = 1;
}
```

📌 *Efecto:* Finaliza el bucle principal (loop()).

6- Modos de Direccionamiento (Cálculo de la dirección efectiva)

```
uint16_t direccionar(uint8_t modo, uint16_t cd) {
    switch (modo) {
        case 0: return cd;              // Directo
        case 1: return mem[cd];         // Indirecto
        case 2: return cd + x;          // Indexado
        case 3: return mem[cd] + x;     // Indirecto+Indexado
    }
    return 0;
}
```

Modo	Descripción	Ejemplo	Resultado
0	Directo	LD 0x20	Usa el valor en mem[0x20]
1	Indirecto	LD (0x20)	Usa el valor en mem[mem[0x20]]
2	Indexado	LD 0x20(X)	Suma X al operando
3	Indirecto + Indexado	LD (0x20)(X)	Doble indirección

En los modos inmediatos (LD #, SUB #), el campo CD **es directamente el dato**, no una dirección.

7- Conclusión General

Este emulador demuestra cómo **el hardware puede ser representado con software**, respetando la estructura y lógica interna de una CPU:

Elemento del Hardware	Equivalente en C
Memoria RAM	uint16_t mem[4096]
Registros	Variables globales
UC (Control Unit)	Bucle loop()
ALU	Conjunto de funciones modulares
Bus de Datos/Direcciones	Parámetros de funciones y arrays

El diseño modular, el uso de **punteros a funciones** y el **modelo de direccionamiento** permiten comprender el flujo interno de un procesador, desde la lectura de instrucciones hasta su ejecución.

Resumen Visual del Ciclo de Ejecución:

1 **Fetch** → lee instrucción desde mem[pc]
2 **Decode** → separa opcode, modo, registro, datos
3 **Execute** → ejecuta iset[op](reg, data)
4 **Update** → incrementa pc o salta según condición

Así, el emulador Simplez-13 transforma la teoría de arquitectura de computadores en un entorno de experimentación completamente funcional y comprensible.

Nuestro programa se precarga en el int main() de main.c, en esencia marcamos las instrucciones a seguir en ensamblador y el emulador las cumple:

```
int main() {

    // --- Carga del Programa en Memoria ---
    uint16_t example_program[] = {
        // Dirección | Contenido | Ensamblador     | Explicación
        /* 0x000 */  0xA87,     // LD.A, #7        ; AC = 7
        /* 0x001 */  0xB94,     // LD.X, #20       ; X = 20


                                (decimal,


                                  0x14)

        /* 0x002 */  0x014,     // ST.A, /20       ; mem[20]=AC


                                (o sea, 7)

        /* 0x003 */  0xA80,     // LD.A, #0        ; AC = 0
        /* 0x004 */  0x494,     // ADD.A, /0[.X]   ; AC = AC +


                                 mem[0 + X]
```

```
                        -> AC = 0 +

                         mem[20] ->

                          AC = 7
/* 0x005 */  0x44A,      // ADD.A, [/10]     ; AC = AC +

                       mem[mem[10]]

                         -> AC = 7 +

                       mem[11] ->

                          AC = 7 + 8 =

                          15
/* 0x006 */  0x4D4,      // ADD.A, [/10][.X]; AC = AC +

                          mem[mem[10]

                          + X] -> AC =

                                                        =
                                                  114
                          15 + mem[11

                       + 20] -> AC

                       = 15 +

                       mem[31] ->
          AC = 15 + 99
/* 0x007 */  0xE00,      // HALT             ; Detiene la
```

```
                                máquina


        // --- Zona de Datos ---
        /* 0x008 */  0x000,
        /* 0x009 */  0x000,
        /* 0x00A */  0x00B,      // Puntero para ADD indirecto

                                          (apunta a la dir 11)
        /* 0x00B */  0x008,      // Valor 8



      // ... (saltamos a la dirección

                                 //      31) …

        [31] = 0x063 // Valor 99 (en hex)
};
```

8- Apartado Adicional: El Depurador y la Interfaz del Terminal

El **depurador** es nuestra herramienta de observación. En una CPU real, no podríamos "ver" nada de lo que pasa internamente: el procesador trabaja en silencio. Pero en el emulador, hemos añadido una capa visual que imprime toda la información del estado de la máquina.

Esto se logra con dos funciones clave del lenguaje C: printf() (para mostrar información en pantalla) y getchar() (para pausar la ejecución entre instrucciones). Así, cada vez que la CPU emulada ejecuta una instrucción, nosotros podemos ver el antes y el después en el terminal.

1. Anatomía de la Salida del Terminal:

Cada instrucción genera **dos líneas**: una que muestra lo que ocurre *antes* de ejecutarla (análisis) y otra que muestra el resultado *después* de ejecutarla.

Primera línea: "ANTES" (Análisis de la Unidad de Control):

```
DEBUG: PC:004 | Inst:494 | OP: ADD | REG: AC | M:2, CD:14 | EA: 014
(Lee de Mem: 007)
```

PC:004 | Inst:494 → La CPU está en la dirección de memoria 004 y ha leído el valor 0x494, que representa la instrucción actual.

OP: ADD | REG: AC | M:2, CD:14 → Se ha identificado que la operación es una suma (ADD), que trabaja con el registro AC, y que usa el modo de direccionamiento "2" con el código de dirección 14.

EA: 014 (Lee de Mem: 007) → Se calcula la **dirección efectiva (EA)**, es decir, dónde están los datos que se van a usar. Aquí, el emulador incluso nos dice que el valor en esa dirección de memoria es 7.

Segunda línea: "DESPUÉS" (Resultado de la Ejecución):

```
-> ESTADO: PC:005, AC:007 (7), X:014 (20), Z_flag:0, H_flag:0
```

(Muestra cómo ha cambiado el estado del sistema tras ejecutar la instrucción)

PC:005 → El contador de programa avanza a la siguiente instrucción.

AC:007 (7) → El acumulador ahora contiene el resultado de la operación (antes era 0, ahora 7).

X:014 (20) → El registro índice no se modificó.

Z_flag:0 → Como el resultado no es cero, la bandera de cero está en 0.

De esta forma, el depurador convierte cada ciclo del procesador en una pequeña historia que podemos seguir paso a paso.

2. Cómo Personalizar la Salida del Depurador (main.c)

Puedes cambiar lo que se muestra en pantalla o incluso desactivar la pausa y la depuración. Todo se hace desde el archivo `main.c`.

A. Cambiar el texto del depurador: Edita la función `print_debug()` para modificar los textos que aparecen. Por ejemplo:

```
printf(" REG: %s | M:%d, CD:%02X | EA: %03X (Valor Leído: %03X)",
reg_names[reg], dirm, cd, data, mem[data]);
```

B. Activar el modo rápido (sin pausas): Si no quieres pulsar Enter en cada paso, comenta la línea `getchar();` al final del bucle.

```
// getchar(); // Desactivar para ejecución continua
```

C. Activar el modo silencioso (sin depuración): Para ver sólo el resultado final del programa, comenta la línea donde se llama a `print_debug()`.

```
// print_debug(inst, op, reg, dirm, cd, data); // Desactivar
depuración
```

El depurador es, en esencia, la voz del emulador. Gracias a él podemos entender lo que ocurre en cada ciclo y observar cómo cada instrucción afecta el “estado interno” de nuestra CPU simulada. Esta visión detallada convierte al Simplez13 en una herramienta ideal para aprender arquitectura de computadores desde dentro, de forma clara y comprensible.