Daniel Ángulo Suescún, Arkai Julián Ariza, Miguel Ángel Castro

dangulos@unal.edu.co, arjarizami@unal.edu.co, miacastroco@unal.edu.co

Resumen

*Este documento plantea las pautas generales para comprender el uso y funcionamiento del simulador del microprocesador Z80.*

MANUAL del Simulador z80

Incluye Manual Técnico y de Usuario

*Octubre 2018*

*Universidad Nacional de Colombia Sede Bogotá*

### Índice

## Introducción

1. **Objetivo del Manual**
2. **Manual de Usuario**
   1. **Estructura de la interfaz**
   2. **Funcionamiento de la Interfaz**
   3. **Escenarios de Prueba**
      1. **Escenario Uno**
      2. **Escenario Dos**
      3. **Escenario Tres**
3. **Manual Técnico**
   1. **Análisis General del Código**
      1. **Clase Estática Z80**
      2. **Funciones Menores**
      3. **Ejecución de la Simulación**
      4. **Conexión al GUI**
   2. **Posibles Mejoras y Bugs**

simulador cpu z80 V. 1.0.  
Manual técnico y de usuario

Daniel Á. Suescún, Arkai J. Ariza, Miguel Á. Castro

|  |  |
| --- | --- |
|  | Este documento plantea las pautas generales para comprender el uso y funcionamiento del simulador del microprocesador Z80. |

# Introducción

Este manual está orientado al entendimiento y uso de un simulador de la CPU Z80. La CPU Z80, también conocida como Zilog Z80 fue creado en 1976 como un procesador Little-endian de 8 bits. Es un híbrido entre la arquitectura de acumulador y la de registros de propósitos generales. Hoy en día existen multitud de variaciones con base en este famoso microprocesador de finales de los 70s.

Se va a comenzar por establecer el objetivo del manual. Posteriormente, se va a dar inicio al manual de usuario donde se va dar a conocer la estructura y el funcionamiento del simulador mediante su interfaz, y se van a dejar tres escenarios de prueba en donde se evidencia los diferentes usos del simulador. Finalizados los escenarios de prueba se va a desarrollar el manual técnico donde se hará un análisis general del código que compone el simulador viendo sus partes, tales como la clase Z80, las funciones menores, la ejecución de la simulación y su conexión al GUI (Graphical User Interface). Finalmente, se van a plantear posibles mejoras junto a los errores conocidos en el programa.

# Objetivo

Establecer un instrumento de apoyo para utilizar y entender el funcionamiento del simulador del microprocesador Z80.

# Manual de usuario

## Estructura de la Interfaz

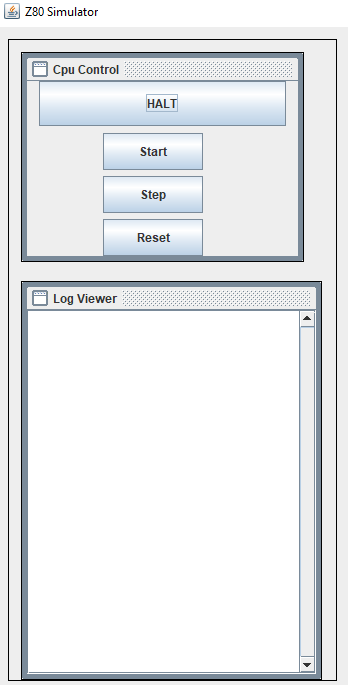
El simulador Z80 cuenta con una sencilla interfaz compuesta principalmente en 2 partes, en primera instancia el CPU Control en donde se encuentran los botones HALT, Start, Step y Reset. En segunda, el Log Viewer en donde se visualiza la información importante de las operaciones que realiza el procesador. Esto se puede apreciar en la siguiente imagen:

Figure 1: Z80 Simulator Interface

## Funcionamiento de la Interfaz

En primera instancia vale aclarar que el simulador recibe un archivo de texto, *Memory.txt*, el cual contiene todos los datos para que la CPU realice todas las operaciones necesarias.

Dicho esto, se va a explicar las posibilidades funcionamiento de cada parte de la interfaz.

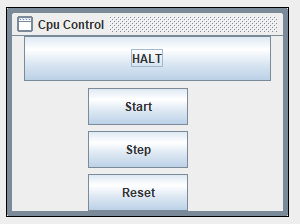


Figure 2: CPU Control Interface

* **Start:** Este botón inicia el programa en *Memory.txt*, y lo ejecuta con normalidad, es importante tener en cuenta que seleccionar el botón Step modificará la ejecución por defecto de Start y por tanto no debe oprimirse antes si se desea que el programa simplemente se ejecute. Cuando Start se ejecuta, en la sección de Log Viewer se verán los resultados de las operaciones.
* **Step:** Este botón se utiliza para ejecutar cada operación individualmente y que, por tanto, no se ejecute todo el programa al iniciar las operaciones. Para ejecutar de esta manera el programa primero se tiene que oprimir el botón Step, y luego oprimir el botón Start, esto iniciará el programa hasta la primera instrucción, posteriormente se tendrá que volver a seleccionar el botón Step para avanzar cada operación, y continuar en esa sucesión hasta terminar el programa. Actualmente, el simulador no cuenta con la capacidad de terminar todo el programa automáticamente una vez entrado en Step, es decir, si se entra en Step habrá que ejecutar operación por operación hasta terminar el programa. No obstante, si es posible pasar de la ejecución automática del programa hacía Step, en otras palabras, mientras el programa se está ejecutando normalmente, resultado de sólo oprimir Start, se puede seleccionar Step para pasar a ejecutar operación por operación.
* **Reset:** Este botón vacía la sección del Log Viewer y termina, sin importar el momento, la ejecución del programa actual, lo que permite volver a ejecutarlo con Start o Step/Start.
* **HALT:** Este botón permite que durante la ejecución de un programa, se lleve directamente al fin de este. Es decir, que mientras un programa se está ejecutando mostrando sus resultados en el Log Viewer se lleva directamente desde ese preciso momento hacia el fin sin terminar lo que falte del programa. Se puede observar en el Log Viewer el resultado que diría *End*.

## Escenarios de Prueba

#### Escenario Uno

#### Escenario Dos

#### Escenario Tres

# Manual Técnico

## Análisis General del Código

En esta sección se mostrará las secciones más relevantes de código que pueden dar una visión general de la construcción del simulador y, posteriormente, se explicara en qué consisten esas líneas de código.

#### Clase Estática z80

static class z80 {

String Address;

String dataBus;

boolean M1, MREQ, IORQ, RD, WR, RFSH;

boolean HALT, WAIT, INT, NMI, RESET;

boolean BUSRQ, BUSACK;

boolean CLK, V5, GROUND;

String Flags = "00000000";

int A, F;

int B, C;

int D, E;

int H, L;

//Index Pointers

int IX;

int IY;

public void setF() {

this.F = binToDec(this.Flags);

}

public void checkAcc() {

if (this.A == 0) {

this.Flags = this.Flags.substring(0, 1) + "1" + this.Flags.substring(2);

} else {

this.Flags = this.Flags.substring(0, 1) + "0" + this.Flags.substring(2);

}

if (this.A < 0) {

this.Flags = "1" + this.Flags.substring(1);

} else {

this.Flags = "0" + this.Flags.substring(1);

}

}

public void setFZero() {

this.Flags = this.Flags.substring(0, 2) + "000000";

}

}

Aquí se establecen algunas de las características principales del z80, entre estas, en las primeras líneas podemos observar la dirección, el bus de datos, el control de sistema, el control de CPU, el control de bus de CPU, los registros del A al F y los punteros de índice (IX IY). Luego podemos ver las siguientes funciones:

* setF - Da a F el valor de la cadena Flags (el registro F).
* checkAcc() - Revisa el valor del acumulador y actualiza las banderas.
* setFZero - pone en 0 los valores relacionados a la operación anterior.

#### Funciones Menores

static String hexToBin(String hex) {

String binAddr = Integer.toBinaryString(Integer.parseInt(hex, 16));

while (binAddr.length() < 8) {

binAddr = "0" + binAddr;

}

return binAddr;

}

static int hexToDec(String hex) {

String binary = hexToBin(hex);

int decimal = Integer.parseInt(binary.substring(1), 2);

if (binary.charAt(0) == '1') {

decimal -= 128;

}

return decimal;

}

static int binToDec(String binary) {

int decimal = Integer.parseInt(binary.substring(1), 2);

if (binary.charAt(0) == '1') {

decimal -= 128;

}

return decimal;

}

static String decToBin(int n) {

String binary = Integer.toBinaryString(n);

while (binary.length() < 8) {

binary = "0" + binary;

}

if (binary.length() > 8) {

binary = binary.substring(24);

}

return binary;

}

static String decToHex(int n) {

String binary = decToBin(n);

int decimal = Integer.parseInt(binary, 2);

String hex = Integer.toString(decimal, 16);

if (hex.length() < 2) {

hex = "0" + hex;

}

hex = hex.toUpperCase();

return hex;

}

Los anteriores métodos pasan los datos de uno a otro de los posibles formatos:

* Bin: Valor binario que se guarda en un String de tamaño 8.
* Hex: Valor hexagecimal que se guarda en un string y siempre mantiene un largo de 2 caracteres, incluso para negativos. En este formato están los valores de la memoria para simular una memoria con una facilidad mayor al binario.
* Dec: Valor decimal guardado en un int para mayor facilidad de operación, en este formato están los registros por facilidad. Los valores se encuentran entre 127 y -128.

#### Ejecución de la Simulación

Dada la gran extensión de esta sección sólo se mostrará una pequeña parte para ilustrar el funcionamiento general. Partes del código que no se muestren se van a expresar como ***(code ..)***.

public static void runSimulation(){

***(code ..)***

String memorytxt = "";

String req;

String req5;

String req2;

String req8;

String pos1;

String pos2;

String temphl;

String tempidx;

int size;

int index;

boolean end = false;

state = "";

En esta primera sección de arriba observamos que memorytxt es la posición que ayuda a guardar los datos del txt a el arreglo que simula la memoria. La variable req ayuda a guardar el registro actual completo, req5 ayuda a guardar el registro actual de la posición 0 a 5, req2 ayuda a guardar el registro actual de la posición 0 a 2, req8 ayuda a guardar el registro actual de la posición 5 a 8, pos1 y pos 2 ayudan a guardar el código de un registro cada uno, e index guarda la posición en la memoria que se está leyendo.

z80 z8 = new z80();

String[] Memory = new String[65536];

for (int i = 0; i < 65536; i++) {

Memory[i] = "00";

}

try {

FileReader fr = new FileReader("Memory.txt");

BufferedReader br = new BufferedReader(fr);

String st;

while ((st = br.readLine()) != null) {

memorytxt = memorytxt + st;

}

System.out.println("Memory " + memorytxt);

gui.updateLogText(memorytxt+"\n");

} catch (Exception e) {

System.out.println("Error Uploading File");

gui.updateLogText("Error Uploading File"+"\n");

}

size = 0;

while (memorytxt.length() != 0) {

if (memorytxt.charAt(0) == 'x' || memorytxt.charAt(1) == 'x') {

memorytxt = memorytxt.substring(1);

} else {

Memory[size] = memorytxt.substring(0, 2);

memorytxt = memorytxt.substring(2);

size++;

}

}

index = 0;

En esta sección de arriba, simplemente cargamos Memory.txt, y lo inicializamos en un string y luego en un arreglo para su posterior uso.

while (!end) {

***(code ..)***

req = hexToBin(Memory[index]);

System.out.println(Memory[index] + "(" + index + ")" + ": " + req);

gui.updateLogText(Memory[index] + "(" + index + ")" + ": " + req+"\n");

req5 = req.substring(0, 5);

req2 = req.substring(0, 2);

req8 = req.substring(5);

***(code ..)***

if(reset){

return;

}

if(state!="end"){

if (req5.equals("10000")) {

state = "add";

} else if (req.equals("01110110")) {

state = "end";

} else if (req.equals("11000010")) {

state = "jnz";

} else if (req.equals("11000011")) {

state = "j\*\*";

} else if (req.equals("11001010")) {

state = "jz";

} else if (req.equals("00100001")) {

state = "ldhl";

} else if (req5.equals("10010")) {

state = "sub";

} else if (req5.equals("10001")) {

state = "adc";

} else if (req5.equals("10011")) {

state = "sbc";

} else if (req5.equals("10100")) {

state = "and";

} else if (req5.equals("10101")) {

state = "Xor";

} else if (req5.equals("10111")) {

state = "com";

} else if (req5.equals("10110")) {

state = "or";

} else if (req2.equals("01")) {

state = "ldr";

} else if (req2.equals("00") && req8.equals("110")) {

state = "ldn";

} else {

state = "xxx";

}

}

***(code ..)***

}

En esta sección de arriba se observa que la identificación de los códigos se hace con **if** y no con **switch** por la facilidad que lleva poner los códigos más largos al principio y a los más cortos al final. Cómo los códigos más cortos se identifican todavía incompletos puede haber colisión así que se ponen los códigos que son excepciones al principio para evitar estos casos. Ej: State "end" se puede ver como lr hl -> hl.

Posteriormente, según el estado se ejecuta la operación entonces, *jnz* es un jump si lo que está en el acumulador es negativo o 0; *j\*\** es un jump a la posición de memoria que se especifica; *jz* es un jump si el acumulador es 0; *ldhl* carga en HL la siguiente posición de memoria; *add* suma A con el registro especificado; *sub* resta el registro especificado a A y guarda el resultado en A; *and* es una operación and entre A y el registro especificado, y se guarda en A; *Xor* es una operación xor (o excluyente) entre A y el registro especificado, y se guarda en A; *or* es una operación or entre A y el registro especificado, y se guarda en A; en *com* se resta el registro r a A y altera las flags congruentemente; *ldr* carga en un registro R el valor de un registro R'; *ldn* guarda en el registro R el valor de la siguiente posición de memoria; *adc* suma el carry a A; *sbc* resta el carry a A; finalmente, en *end* el código finaliza la ejecución.

#### Conexión al GUI

public Main() {

initComponents();

***(code ..)***

MouseListener MouseLst = new MouseListener(){

@Override

public void mouseClicked(MouseEvent e){

check = true;

};

***(code ..)***

};

stepButton.addMouseListener(MouseLst);

}

***(code ..)***

static boolean check = false;

En esta primera sección de arriba observamos que se pone un Listener en el botón se Step, esto se hace con el simple objetivo de poder avanzar cada operación por separado, para cumplir esto también se hace uso de la función checkStep() que está más adelante.

private void startButtonActionPerformed(java.awt.event.ActionEvent evt) {

Z80.startSim = true;

}

private void haltButtonActionPerformed(java.awt.event.ActionEvent evt) {

Z80.state="end";

}

private void resetButtonActionPerformed(java.awt.event.ActionEvent evt) {

jTextArea1.setText("");

Z80.reset = true;

}

private void stepButtonActionPerformed(java.awt.event.ActionEvent evt) {

if(Z80.stepMode == false){

Z80.step = true;

}

Z80.stepMode = true;

}

public void updateLogText(String x){

jTextArea1.append(x);

}

public void checkStep(){

if(check==true){

Z80.step = false;

}

check=false;

}

}

En esta sección de arriba observamos las acciones de los botones de la interfaz, el botón de Start permite cambiar la variable startSim que da paso al inicio de la ejecución del programa; el botón halt lleva directamente a la terminación del programa mediante el cambio de state a *end*; el botón reset limpia la sección de Log Viewer y modifica la variable que permite que el sistema conozca que se debe dar un reset; el botón Step permite cambiar las variables que permiten iniciar el programa paso por paso. Finalmente, está la función updateLogText() que permite que se actualice la información en el Log Viewer.

## Posibles Mejoras Y Bugs

|  |  |
| --- | --- |
| Posibles Mejoras | Tiempo Estimado (Días) |
| Permitir Pasar de ejecutar el programa paso a paso a automático | 2 |
| Aumentar la cantidad de operaciones posibles | 5 |
| **Bugs Conocidos** | |
| El Reset no funciona correctamente después de algunos intentos seguidos haciendo reset. | |