Trabajo Práctico 1 — Conjunto de instrucciones MIPS

[66.20] Organización de Computadoras
Curso 2
Primer cuatrimestre de 2021

Alumnos	Padrón	Email
ARRACHEA, Tomás	104393	tarrachea@fi.uba.ar

Índice

1. Enunciado

66:20 Organización de Computadoras
Trabajo práctico 1: conjunto de instrucciones MIPS

1. Objetivos

Familiarizarse con el conjunto de instrucciones MIPS32 y el concepto de ABI[1], escribiendo un programa portable que resuelva el problema descripto en la sección 6.

2. Alcance

Este trabajo práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

3. Requisitos

El trabajo deberá ser entregado personalmente, en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes.

Además, es necesario que el trabajo práctico incluya (entre otras cosas, ver sección 9), la presentación de los resultados obtenidos, explicando, cuando corresponda, con fundamentos reales, las causas o razones de cada resultado obtenido.

El informe deberá respetar el modelo de referencia que se encuentra en el grupo, y se valorarán aquellos escritos usando la herramienta TEX / LATEX.

4. Recursos

Usaremos el programa QEmu[1] para simular el entorno de desarrollo de una máquina MIPS corriendo una versión reciente del sistema operativo Debian[2].

[1] Application binary interface

5. Introducción

Los autómatas celulares son un caso de modelo matemático para un sistema dinámico que evoluciona en pasos discretos. Originalmente creados en el contexto de la física computacional, se utilizan en varios otros campos, como la teoría de la computabilidad. Mientras que muchos autómatas celulares, como en el caso del Juego de la Vida de Conway [4], evolucionan en una matriz bidimensional, otros, como los autómatas celulares elementales o unidimensionales, lo hacen en un array unidimensional. Esto nos permite almacenar la evolución en el tiempo del autómata como una segunda dimensión. En los autómatas celulares elementales, las celdas en las que está dividido nuestro universo pueden contener una célula o no, y el estado de cada celda en la iteración i depende del estado de esa celda y sus vecinos inmediatos a izquierda y derecha en la iteración $i - 1$. Por ejemplo, si expresamos el estado de una celda y sus dos celdas adyacentes como dígitos binarios, donde 0 representa una celda vacía y 1 una ocupada, podemos determinar unívocamente el comportamiento de un autómata celular elemental asignando un dígito binario a cada combinación de tres bits, donde 1 representa que una celda con ese contenido y esos vecinos está ocupada en la siguiente iteración, y 0 representa que no. Por ejemplo, la siguiente tabla representa un posible autómata:

111	110	101	100	011	010	001	000
0	0	0	1	1	1	1	0

En este caso, una celda vacía cuyo vecinos izquierdo y derecho estén ocupado y vacío respectivamente pasará a estar ocupada, una celda ocupada cuyo vecino izquierdo esté vacío seguirá ocupada, una celda ocupada sin vecinos seguirá estando ocupada, una celda vacía cuyos vecinos izquierdo y derecho estén vacío y ocupado respectivamente pasará a estar ocupada, y todas las otras celdas quedarán vacías.

5.1. Numeración de Wolfram

Atendiendo a que la fila 2 de la tabla anterior puede verse como un número binario de 8 bits, el matemático Stephen Wolfram propuso denominar las reglas por este número. Así, el autómata definido por la tabla anterior es conocido como “Regla 30”. La regla 30 en particular muestra un comportamiento caótico, como se ve en la figura 1. La regla 126, por otro lado, si partimos de una única celda ocupada nos muestra una aproximación bastante ajustada del triángulo de Sierpinski [7] como se ve en la Figura 2.

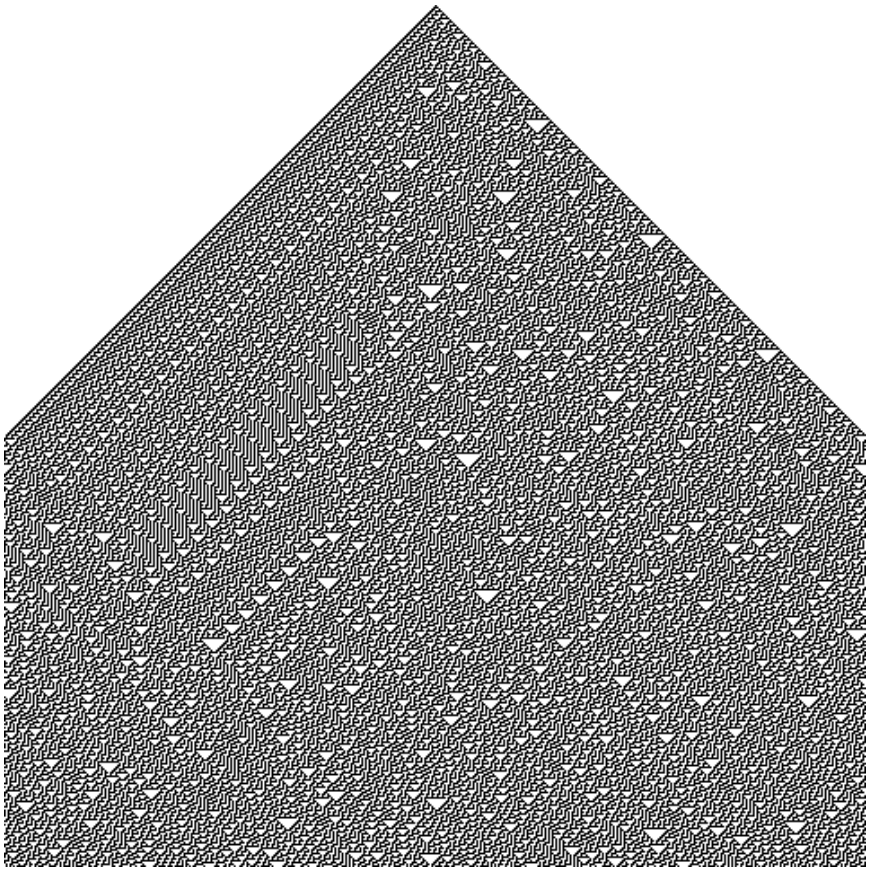

Figura 1: Regla 30 a partir de una celda ocupada en el centro

6. Programa

Se trata de una versión en lenguaje C de un programa que computa autómatas celulares para reglas arbitrarias. El programa recibirá por como argumentos el número de regla R, la cantidad de celdas de una fila N y el nombre de un archivo de texto con el contenido del estado inicial del autómata, y escribirá un archivo .PBM [5] representando la evolución del autómata celular en una matriz de NxN . El archivo de estado inicial debe contener una línea con N dígitos binarios, con 1 representando una celda ocupada y 0 una vacía. De haber errores, los mensajes de error deberán salir exclusivamente por `stderr`.

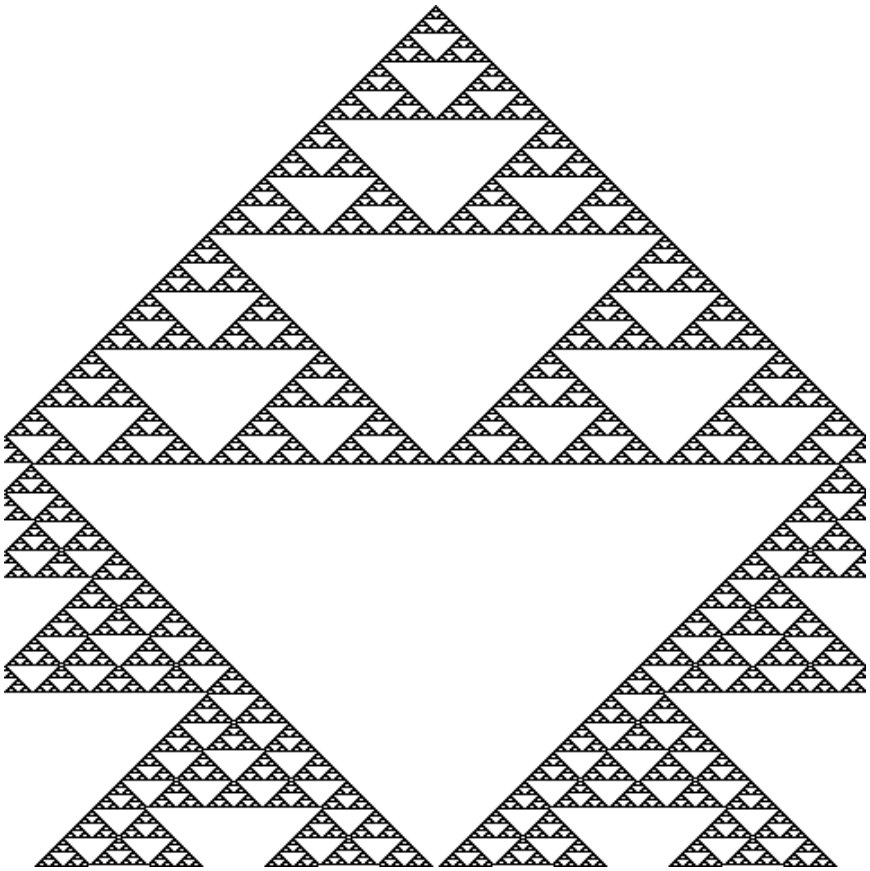

Figura 2: Regla 126 a partir de una celda ocupada en el centro

6.1. Condiciones de contorno

Para calcular los vecinos de las celdas de los extremos de la matriz, lo que hacemos es utilizar la hipótesis del mundo toroidal: La fila $N-1$ pasa a ser vecina de la fila 0, y la columna $N-1$ pasa a ser vecina de la columna 0. Entonces, el vecino superior de la celda $[0, j]$ es el $[N-1, j]$, el vecino izquierdo de la celda $[i, 0]$ es el $[i, N-1]$, y viceversa; de esta manera, nunca nos salimos de la matriz. En este caso, por supuesto, nuestro mundo es en realidad unidimensional y los vecinos superiores e inferiores de una celda no nos interesan.

6.2. Comportamiento deseado

Primero, usamos la opción `-h` para ver el mensaje de ayuda:

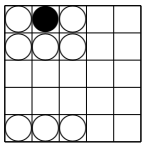

Figura 3: Ejemplo de mundo toroidal: la celda (0,1) y sus vecinos.

```
$ autcel -h
Uso:
  autcel -h
  autcel -V
  autcel R N inputfile [-o outputprefix]
Opciones:
  -h, --help     Imprime este mensaje.
  -V, --version  Da la versión del programa.
  -o  Prefijo de los archivos de salida.
Ejemplos:
 autcel 30 80 inicial -o evolucion
 Calcula la evolución del autómata "Regla 30",
 en un mundo unidimensional de 80 celdas, por 80 iteraciones.
 El archivo de salida se llamará evolucion.pbm.
 Si no se da un prefijo para los archivos de salida,
 el prefijo será el nombre del archivo de entrada.
```

Ahora usaremos el programa para generar una secuencia de estados del autómata.

```
$ autcel 30 80 inicial -o evolucion
Leyendo estado inicial...
Grabando evolucion.pbm
Listo
```

El formato del archivo de entrada es de texto, con N dígitos binarios representando la ocupación de las celdas. Ejemplo: si el archivo `inicial` representa sólo una celda ocupada en el centro de un mundo de 9 celdas, se verá de la siguiente manera:

```
$ cat inicial
000010000
$
```

El programa deberá retornar un error si la cantidad de celdas difiere de N, o si el archivo no cumple con el formato.

7. Implementación

El programa a implementar deberá satisfacer algunos requerimientos mínimos, que detallamos a continuación.

7.1. Portabilidad

Pese a contenter fragmentos en assembler MIPS32, es necesario que la implementación desarrollada provea un grado mínimo de portabilidad.

Para satisfacer esto, el programa deberá proveer dos versiones de `autcel()`, incluyendo la versión MIPS32, pero también una versión C, pensada para dar soporte genérico a aquellos entornos que carezcan de una versión más específica.

7.2. API

Gran parte del programa estará implementada en lenguaje C. Sin embargo, la función `proximo()` estará implementada en assembler MIPS32, para proveer soporte específico en nuestra plataforma principal de desarrollo, Debian/QEmu.

El programa en C deberá interpretar los argumentos de entrada, pedir la memoria necesaria para la matriz de estado A, popular la matriz con los contenidos del archivo de entrada, y computar el siguiente estado para cada celda valiéndose de la siguiente función:

```
unsigned char proximo(unsigned char *a,
                      unsigned int i, unsigned int j,
                      unsigned char regla, unsigned int N);
```

Donde `a` es un puntero a la posición $[0, 0]$ de la matriz, i y j son la fila y la columna respectivamente del elemento cuyos vecinos queremos calcular, y N es la cantidad de filas y columnas de la matriz A. El valor de retorno de la función `proximo` es el valor de ocupación de la celda $[i+1, j]$, o sea el estado de la celda j en la iteración $i+1$. Los elementos de A pueden representar una celda cada uno, aunque para reducir el uso de memoria podrían contener hasta ocho cada uno. Después de computar el siguiente estado para la matriz A, el programa deberá escribir un archivo en formato PBM [5] representando las celdas encendidas con color blanco y las apagadas con color negro [2].

7.3. ABI

El pasaje de parámetros entre el código C (`main()`, etc) y la rutina `proximo()`, en assembler, deberá hacerse usando la ABI explicada en clase:

[2] Si se utiliza sólo un pixel por celda, no se podrá apreciar el resultado a simple vista. Pruebe haciendo que una celda sea representada por grupos de por ejemplo 4x4 pixels.

los argumentos correspondientes a los registros `$a0-$a3` serán almacenados por el *callee*, siempre, en los 16 bytes dedicados de la sección "function call argument area" [3].

7.4. Algoritmo

El algoritmo a implementar es el algoritmo de autómatas celulares unidimensionales[6], explicado en clase.

8. Proceso de Compilación

En este trabajo, el desarrollo se hará parte en C y parte en lenguaje Assembler. Los programas escritos serán compilados o ensamblados según el caso, y posteriormente enlazados, utilizando las herramientas de GNU disponibles en el sistema Debian utilizado. Como resultado del enlace, se genera la aplicación ejecutable.

9. Informe

El informe deberá incluir:

- Este enunciado;
- Documentación relevante al diseño e implementación del programa, incluyendo un diagrama del stack de la función `proximo`;
- Corridas de prueba para una matriz de lado 80, de las reglas 30, 110 y 126, con una celda ocupada en el centro como estado inicial.
- El código fuente completo, en formato digital. Es conveniente ponerlo al final del informe, como apéndice.

10. Mejoras opcionales

- Una versión de terminal, que permita ver en tiempo real la evolución del sistema (y suprima los archivos de salida).
- Un editor de pantalla, de modo texto, a una celda por caracter. Esto permite experimentar con el programa, particularmente combinado con la versión de tiempo real.

11. Fecha de entrega

Primera entrega: Jueves 20 de Mayo de 2021. Revisión: Jueves 27 de Mayo de 2021. Última oportunidad de entrega: Jueves 3 de Junio de 2021.

Referencias

[1] QEmu, `https://www.qemu.org/`.

[2] Debian, the Universal Operating System, `https://www.debian.org/`.

[3] System V application binary interface, MIPS RISC processor supplement (third edition). Santa Cruz Operations, Inc.

[4] Juego de la Vida de Conway, `http://es.wikipedia.org/wiki/Juego_de_la_vida`.

[5] Formato PBM: `http://netpbm.sourceforge.net/doc/pbm.html`

[6] Autómatas celulares elementales: `http://mathworld.wolfram.com/ElementaryCellularAutomaton.html`

[7] Triángulo de Sierpinski: `http://mathworld.wolfram.com/SierpinskiSieve.html`

2. Diseño y detalles de implementación

El programa se implementó en C a partir de una matriz dinámica de chars de tamaño n*n. Se carga la primera fila con el contenido inicial del autómata, proporcionado por el archivo inicial. Luego, se accede a cada posición de la matriz y se asigna el valor correspondiente del autómata, calculado por la función *proximo*, que accede a la posición a partir de un puntero al heap y encapsula la lógica de cálculo del autómata.

En el módulo implementado en MIPS, se decidió no guardar los valores de las variables de los registros temporales en el Stack. De esta manera, se logra un rendimiento más rápido, a costa de perder la posibilidad de hacer un backtracing. Esto impide que al momento de debuggear el programa, se pueda acceder al valor de esas variables. Sin embargo, considero que no es necesario para este programa.

3. Stack de la función *proximo*

Diagrama del manejo del stack de la función *proximo*, necesario para el manejo de la ABI:

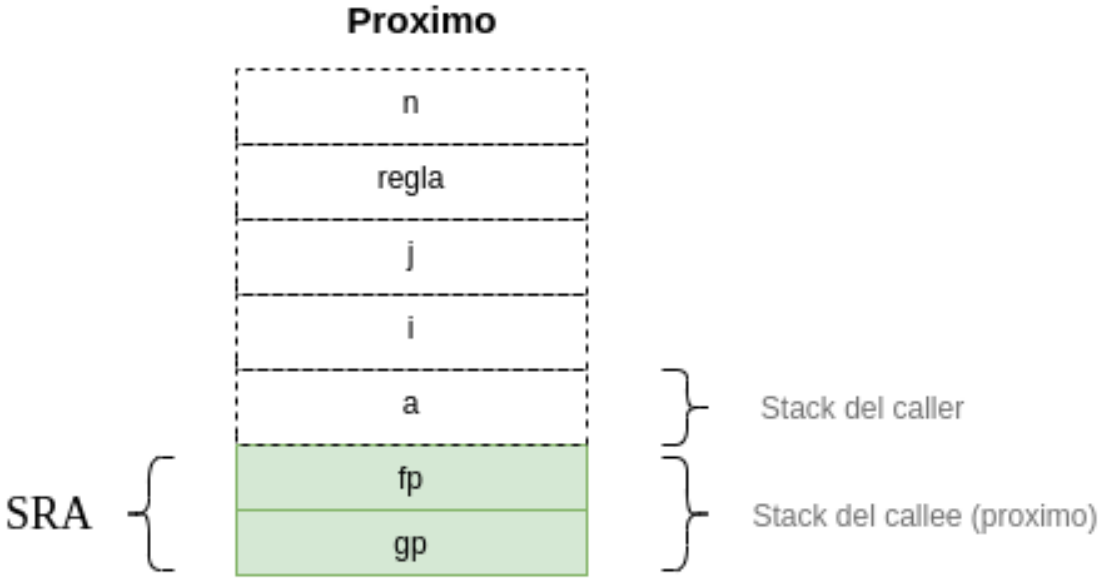

Figura 1: Stack de *proximo*.

4. Ejemplos de ejecución

Se hicieron corridas de prueba para una matriz de lado 80, de las reglas 30, 110 y 126, con una celda ocupada en el centro como estado inicial. Estos fueron los resultados:

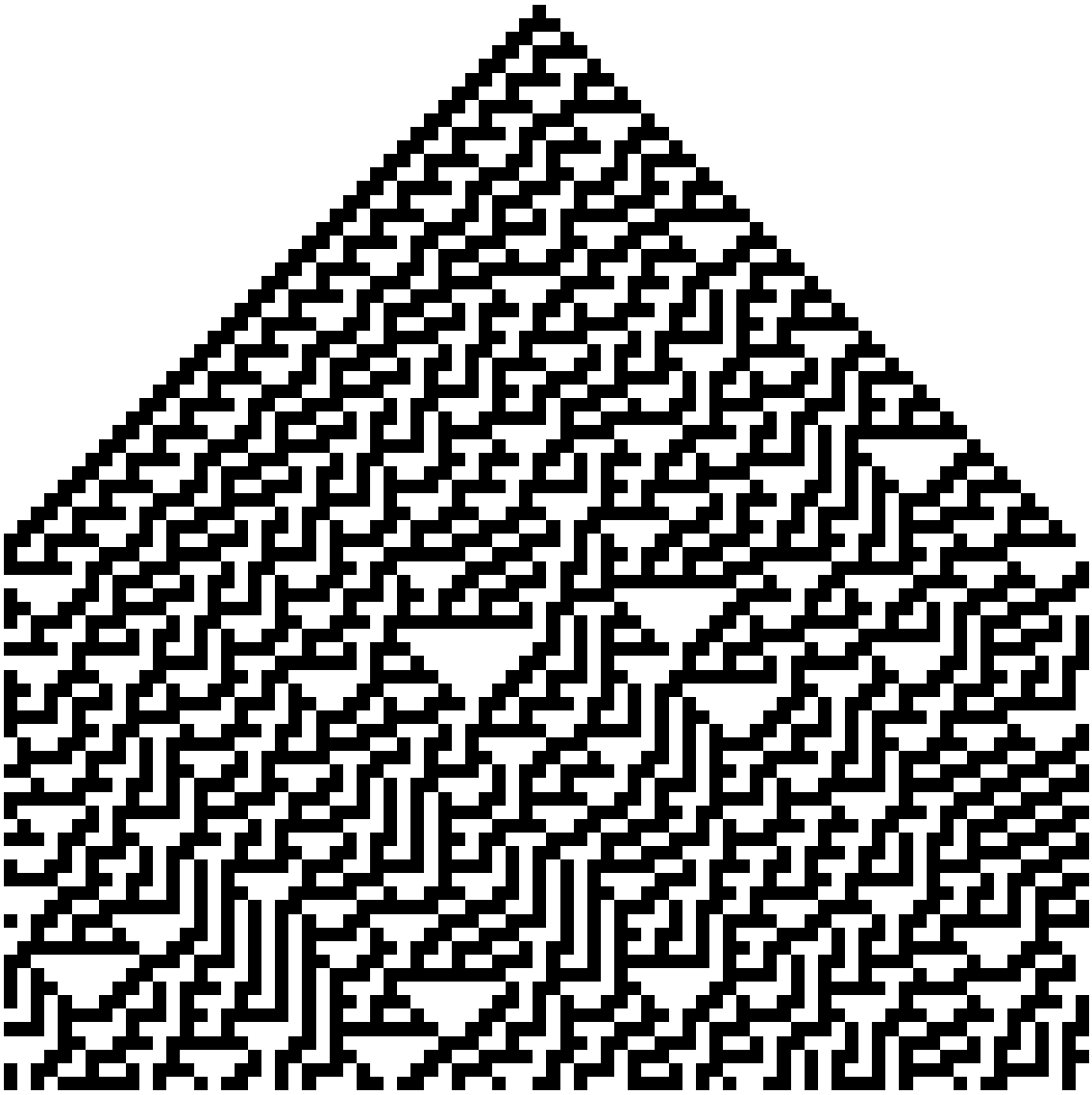

Figura 2: Corrida con regla 30.

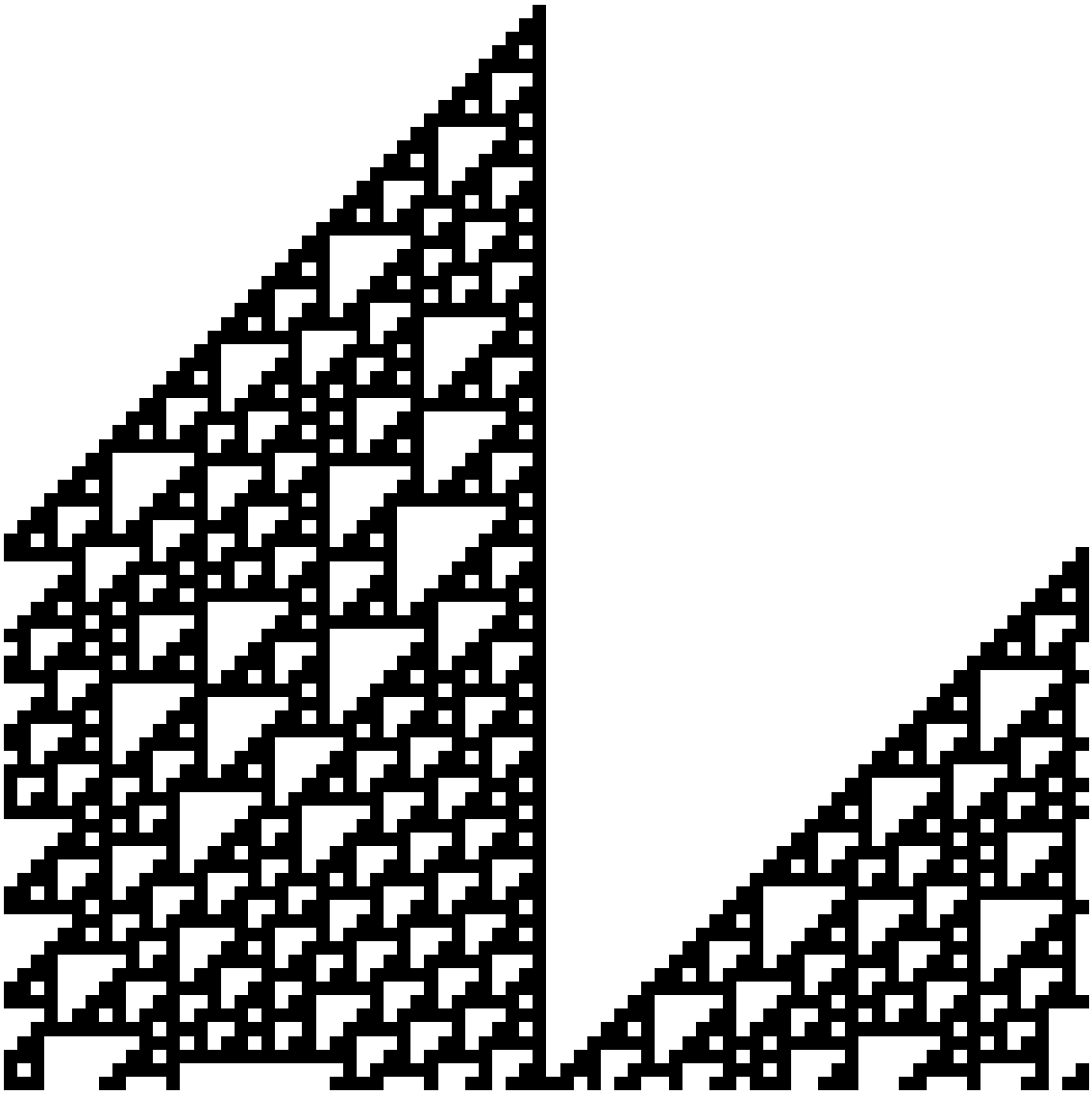

Figura 3: Corrida con regla 110.

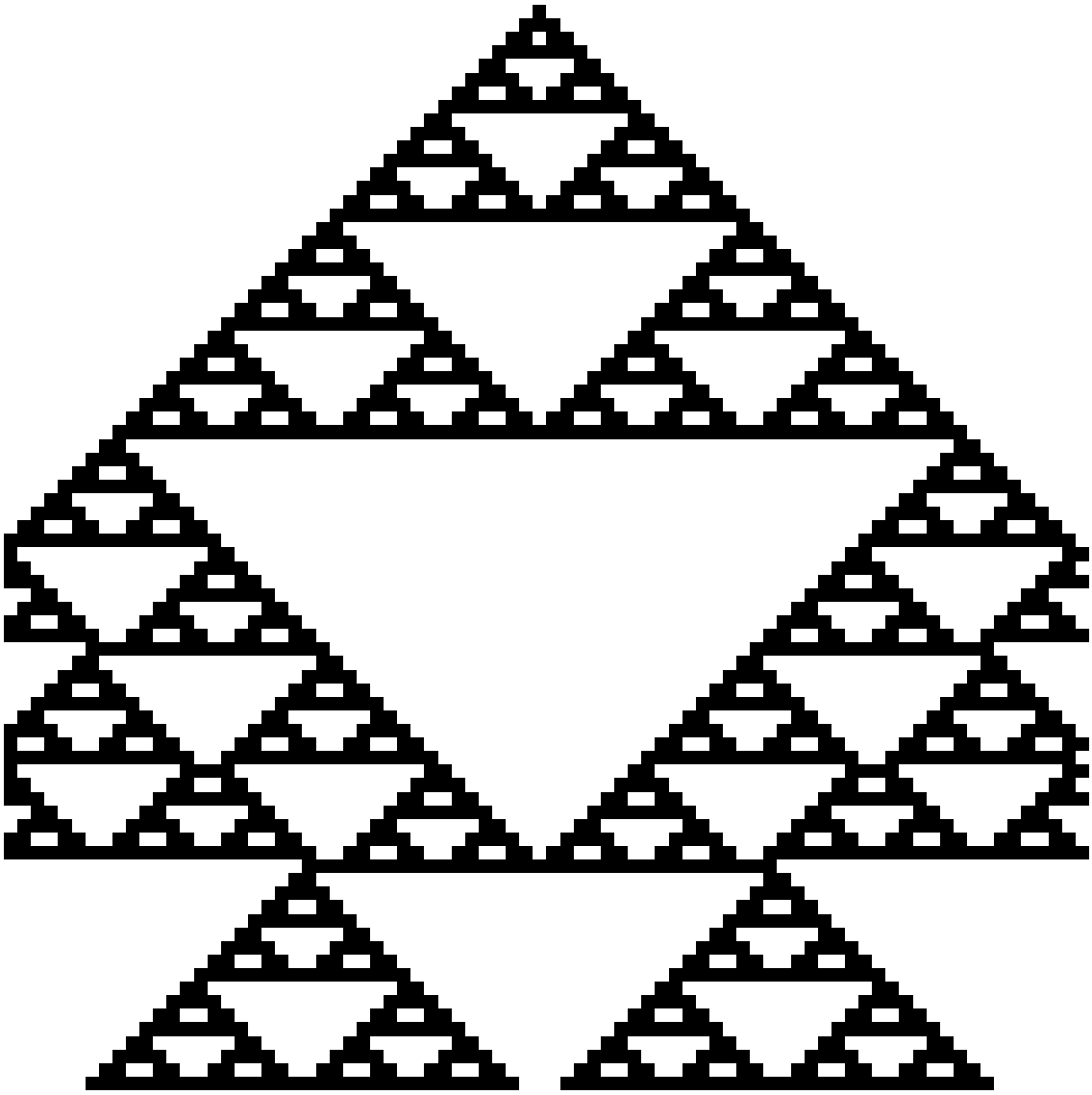

Figura 4: Corrida con regla 126.

5. Apéndice

Código fuente.

5.1. autcel.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>

const char* VERSION_ACTUAL = "1.2";
const int MULTIPLICADOR_PIXELS = 4;
```

```
extern unsigned char proximo(unsigned char* a, unsigned int i, unsigned int j, unsigned char regl
void imprimir_ayuda();
void imprimir_version();
int cargar_inicio(char* nombre_inicial, unsigned char** estados, int n);
void guardar_fila(FILE* salida, unsigned char* estados, int n);
FILE* inicializar_pbm(char* nombre, int n);
void generar_automata(FILE* salida, char* nombre_inicial, char regla, int n);

int main(int argc, char* argv[]) {
FILE* salida = NULL;
int c;
char* nombre_salida = "evolucion.pbm";

while ((c = getopt(argc, argv, "hVpo:")) != -1)
switch (c) {
case 'h':
imprimir_ayuda();
    return 0;
case 'V':
imprimir_version();
    return 0;
case 'p':
salida = stdout;
break;
case 'o':
    nombre_salida = optarg;
    break;
case '?':
    if (optopt == 'o')
fprintf (stderr, "La opción -%c requiere un argumento.\n", optopt);
    else if (isprint (optopt))
     fprintf (stderr, "Opción desconocida `-%c'.\n", optopt);
    else
     fprintf (stderr,"Caracter desconocido `\\x%x'.\n", optopt);
    return 1;
default:
    abort ();
    }

    int index = optind;

    if (argc < 4) {
perror("Error: los argumentos son inválidos.");
return 1;
    }

int regla = atoi(argv[index++]);
    int n = atoi(argv[index++]);
char* nombre_inicial = argv[index++];

    if (salida == NULL) {
salida = inicializar_pbm(nombre_salida, n);
}
```

```
if (regla > 255 || regla <= 0 || n <= 0) {
perror("Error: los argumentos son inválidos.");
fclose(salida);
return 1;
}

generar_automata(salida, nombre_inicial, (char)regla, n);

fclose(salida);
return 0;
}

FILE* inicializar_pbm(char* nombre, int n) {
FILE* file = fopen(nombre, "w");
if (!file){
perror("Error al crear el archivo de salida.");
return 0;
    }

fprintf(file, "P1\n# feep.pbm\n%i %i\n", n*MULTIPLICADOR_PIXELS, n*MULTIPLICADOR_PIXELS);

return file;
}

void generar_automata(FILE* salida, char* nombre_inicial, char regla, int n) {
unsigned char* estados[n];
for (int i = 0; i < n; i++)
estados[i] = malloc(n*sizeof(char));

if (cargar_inicio(nombre_inicial, estados, n) != 0) {
return;
}

guardar_fila(salida, estados[0], n);
for (unsigned int i = 0; i < n-1; i++) {
for (unsigned int j = 0; j < n; j++) {
estados[i+1][j] = proximo((unsigned char*)estados, i, j, regla, n);
}
guardar_fila(salida, estados[i+1], n);
}

for (int j = 0; j < n; j++)
free(estados[j]);
}

void imprimir_ayuda(){
printf(
" Uso:\n"
" autcel -h\n"
" autcel -V\n"
" autcel R N inputfile -s\n"
" autcel R N inputfile [-o outputprefix]\n"
"\n Opciones:\n"
" -h Imprime este mensaje.\n"
```

```
" -V  Da la versión del programa.\n"
" -s  Imprime por consola la salida del programa, en lugar de guardarlo en el archivo de salida.\
" -o  Prefijo de los archivos de salida.\n"
"\n Ejemplos:\n"
" -> autcel 30 80 inicial -o evolucion\n"
" Calcula la evolución del autómata 'Regla 30', en un mundo unidimensional de 80 celdas, por 80 i
" El archivo de salida se llamará evolucion.pbm.\n"
" Si no se da un prefijo para los archivos de salida, el prefijo será el nombre del archivo de en
}

void imprimir_version(){
printf("Versión actual: %s\n", VERSION_ACTUAL);
}

int cargar_inicio(char* nombre_inicial, unsigned char** estados, int n) {
FILE* archivo_inicial = fopen("inicial", "r");
if (!archivo_inicial){
perror("Error al abrir el archivo inicial.");
return 2;
}

char numero;
for (int j = 0; j < n; j++){
numero = getc(archivo_inicial) - '0';
if (numero == EOF){
fclose(archivo_inicial);
perror("Error: el archivo inicial tiene celdas de menos.");
return -1;

} else if (numero != 0 && numero != 1){
fclose(archivo_inicial);
perror("Error: el archivo inicial no cumple con el formato.");
return -1;
}
estados[0][j] = numero;
}

if (fscanf(archivo_inicial, "%s", &numero) != EOF){
perror("Error: el archivo inicial tiene celdas de sobra.");
return 1;
}

fclose(archivo_inicial);

return 0;
}

void guardar_fila(FILE* salida, unsigned char* estados, int n){
for (int k = 0; k < MULTIPLICADOR_PIXELS; k++) {
for (int j = 0; j < n; j++) {
for (int i = 0; i < MULTIPLICADOR_PIXELS; i++) {
fprintf(salida, "%i ", estados[j]);
}
}
```

```
fprintf(salida, "\n");
}
}
```

5.2. proximo.c

```
extern unsigned char proximo(unsigned char** a, unsigned int i, unsigned int j, unsigned char reg
char previo, central, siguiente;
int posicion;

if (j == 0) {
previo = a[i][n-1];
siguiente = a[i][j+1];
} else if (j == n-1) {
previo = a[i][j-1];
siguiente = a[i][0];
} else {
previo = a[i][j-1];
siguiente = a[i][j+1];
}
central = a[i][j];

posicion = previo << 2 | central << 1 | siguiente << 0;

return (1 & (regla >> posicion));
}
```

5.3. proximo.S

```
.text
.align 2
.globl proximo
.ent proximo

proximo:
subu $sp, $sp, 24 # 2 words (SRA) + 4 words (LTA) + 0 words (ABA) = 24 bytes

# SRA
sw $gp, 20($sp)
sw $fp, 16($sp)
move $fp, $sp
#guardo los argumentos en el stack del callee
#$a0 = a
#$a1 = i
#$a2 = j
#$a3 = regla
sw $a0, 24($fp)
sw $a1, 28($fp)
sw $a2, 32($fp)
sw $a3, 36($fp)

# LTA
#$t0 = posicion
#$t1 = digito1
```

```
#$t2 = digito2
#$t3 = digito3
#t$4 = n
lw $t4, 40($fp) #carga n desde el stack del callee

#programa:
sll $a1, $a1, 2 # i * 4, son punteros de 4 bytes
add $a0, $a0, $a1  # $a0 = a + i
lw $a0, 0($a0)
add $a0, $a0, $a2 # $a0 = a[i] + j

#digito2 = estados[i][j]
lb $t2, 0($a0)

##if (j == 0)
beq $zero, $a2, lim_inferior
##if (j == n-1)
addi $t4, $t4, -1
beq $t4, $a2, lim_superior
##else
#digito1 = a[i][j-1]
#digito3 = a[i][j+1]
lb $t1, -1($a0)
lb $t3, 1($a0)
j exit

lim_inferior:
#digito1 = a[i][n-1]
#digito3 = a[i][j+1]
lb $t3, 1($a0)
addu $a0, $a0, $t4
lb $t1, ($a0)
j exit
lim_superior:
#digito1 = a[i][j-1]
#digito3 = a[i][0]
lb $t1, -1($a0)
subu $a0, $a0, $a2
lb $t3, ($a0)
j exit

exit:

#digito1 = digito1 << 2
#digito2 = digito2 << 1
sll $t1, $t1, 2
sll $t2, $t2, 1

#posicion = digito1 << 2 | digito2 << 1 | digito3 << 0
or $t0, $t1, $t2
or $t0, $t0, $t3

#regla >> posicion
srlv $a3, $a3, $t0
```

```
#$v0 = regla & 1.
andi $v0, $a3, 1

#liberar Stack
    lw $fp, 16($sp)
    lw $gp, 20($sp)
    addiu $sp, $sp, 24

jr $ra
.end proximo
```