Trabajo práctico #1: conjunto de instrucciones MIPS

Santiago Fernandez, *Padrón Nro. 94.489*
fernandezsantid@gmail.com

Pablo Rodrigo Ciruzzi, *Padrón Nro. 95.748*
p.ciruzzi@hotmail.com

Horacio Martinez, *Padrón Nro. 94.926*
hmk142@hotmail.com

2do. Cuatrimestre de 2015
66.20 Organización de Computadoras – Práctica Jueves
Facultad de Ingeniería, Universidad de Buenos Aires

24 de Septiembre, 2015

Resumen

Este trabajo práctico trata de una versión en lenguaje C, de un programa que computa autómatas celulares para reglas y estados iniciales arbitrarios. Además, se hizo un version en Assembly de una función para familiarizarse con el conjunto de instrucciones MIPS32 y el concepto de ABI.

Índice

1. Introducción

Un autómata celular es un modelo matemático para un sistema dinámico que evoluciona en pasos discretos; en este trabajo desarrollamos el autómata no trivial más simple que consiste en una retícula unidimensional de células que sólo pueden tener dos estados (0 o 1), con un vecindario constituido, para cada célula, de ella misma y de las dos células adyacentes ($2^3 = 8$ configuraciones posibles).

Existen $2^8 = 256$ modos de definir cuál ha de ser el estado de una célula en la generación siguiente para cada una de estas configuraciones. Estos modos se codifican según un número (llamado de regla) propuesto por Stephen Wolfram.

Dado un número de regla y un estado inicial de N celdas, este programa se encarga de calcular los siguientes N estados para dicha configuración y escribirlos en una imagen de formato pbm.

2. Desarrollo

2.1. Recursos y Portabilidad

Uno de los objetivos del trabajo práctico es poder probar la portabilidad del programa en diferentes entornos. En el enunciado se pedía que el programa se pudiera ejecutar en NetBSD[1] (usando el simulador GXemul[2]) y en la versión de Linux (Knoppix, RedHat, Debian, Ubuntu) usada para correr el simulador. En particular, se lo probó en Ubuntu 14.04. En GXemul se corrió una máquina de arquitectura MIPS cuyo sistema operativo era una versión reciente de NetBSD. La transferencia de archivos entre la máquina host y la guest se hizo mediante *SSH*. Se procedió de la siguiente manera:

Para trabajar con el GXemul se procedió primero creando una nueva interfaz de red (debe crearse cada vez que se inicia el *host* y con permisos de administrador):

hostOS$ sudo ifconfig lo:0 172.20.0.1

Luego se ejecutó el GXemul en modo X:

hostOS$ ./xgxemul -e 3max -d netbsd-pmax.img -x

Una vez ya ingresado con el usuario y la contraseña en la máquina simulada, se creó un túnel reverso para saltear las limitaciones propias del GXemul:

guestOS$ ssh -R 2222:127.0.0.1:22 usuario@172.20.0.1

A partir de ese momento y dejando lo anterior en segundo plano, ya se puede trabajar mediante *SSH* de manera más cómoda:

hostOS$ ssh -p 2222 root@127.0.0.1

2.2. Implementación

La implementación del autómata celular unidimensional[3], tal como se lo explicó en la sección *Introducción*, se hizo íntegramente en el lenguaje de programación C. Luego, también se desarrolló una de las funciones que ya se había implementado en C, en Assembler de una máquina de arquitectura MIPS32. La función a desarrollar en ambos lenguajes fue la función *proximo* la cual, dadas la matriz y la celda que se quiere saber su próximo estado, devuelve este último. Para el pasaje entre C y Assembler se hizo uso de la ABI [4] explicada en la materia.

El resultado de la ejecución del programa es guardado en un archivo PBM[5] el cual muestra el avance del autómata a través de los pasos discretos.

2.3. Compilación

Para compilar el trabajo práctico, realizamos un Makefile para hacer mas sencilla esta tarea.

Para compilar para C: *make c*

Para compilar para MIPS: *make mips*

Finalmente para limpiar los archivos: *make clean*

2.4. Corrida de Pruebas

Teniendo en cuenta un archivo inicial de 80 caracteres de que tiene el siguiente formato:
$ cat inicial
00000000000000000000000000000000000000010000000000000000000000000000000000000000
Y no habiendo ningun otro archivo en la carpeta raíz del proyecto, se hicieron las siguientes pruebas:

- Como primer medida ejecutamos el comando -h para ver la ayuda:
  - ***$ autcel -h***
    Uso:
    autcel -h
    autcel -V
    autcel R N inputfile [-o outputprefix]

    Opciones:

-h, --help	*Imprime este mensaje.*
-V, --version	*Da la version del programa.*
-o	*Prefijo de los archivos de salida.*

    Ejemplo:
    autcel 30 80 inicial -o evolucion
    Calcula la evolucion del automata "Regla 30", en un mundo unidimensional de 80 celdas, por 80 iteraciones.

El archivo de salida se llamara evolucion.pbm.
Si no se da un prefijo para los archivos de salida, el prefijo sera el nombre del archivo de entrada.

- Y luego el comando -V para ver la versión:
  - ***$ autcel -V***
    autcel: version 1.0

- Primero probamos los "casos felices":
  - ***$ ./autcel 30 80 inicial***
    Leyendo estado inicial...
    Calculando los 79 estados siguientes...
    Grabando inicial.pbm
    Listo.

  - ***$ ./autcel 30 80 inicial -o final***
    Leyendo estado inicial...
    Calculando los 79 estados siguientes...
    Grabando final.pbm
    Listo.

- Luego probamos sin enviar ningun tipo de parámetro:
  - ***$ autcel***
    El comando ejecutado no respeta la sintaxis. Para mas ayuda ejecutar el programa con -h o --help.

- Y también cualquier otra idea que se nos ocurrió. Estos son algunos ejemplos:
  - ***$ ./autcel -p***
    El comando ejecutado no respeta la sintaxis. Para mas ayuda ejecutar el programa con -h o --help.

  - ***$ ./autcel 30 80***
    El comando ejecutado no respeta la sintaxis. Para mas ayuda ejecutar el programa con -h o --help.

  - ***$ ./autcel 30 80 primero***
    El archivo de entrada especificado no se ha podido abrir o no existe.

  - ***$ ./autcel -1 80 inicial***
    El numero de regla no es valido. Debe ser un valor numerico entre 0 y 255.

  - ***$ ./autcel 256 80 inicial***
    El numero de regla no es valido. Debe ser un valor numerico entre 0 y 255.

  - ***$ ./autcel hola 80 inicial***
    El numero de regla no es valido. El mismo no contiene un valor

numerico o se ha producido un overflow.

- ***$ ./autcel 30 -8 inicial***
  El numero de celdas e iteraciones ("N") no es valido. Debe ser un valor numerico entre 1 y 10000.

- ***$ ./autcel 30 hola inicial***
  El numero de celdas e iteraciones ("N") no es valido. El mismo no contiene un valor numerico o se ha producido un overflow.

- ***$ ./autcel 30 11111 inicial***
  El numero de celdas e iteraciones ("N") no es valido. Debe ser un valor numerico entre 1 y 10000.

- ***$ ./autcel 30 80 inicial final***
  El comando ejecutado no respeta la sintaxis. Para mas ayuda ejecutar el programa con -h o --help.

- ***$ ./autcel 30 80 inicial -p final***
  El comando ejecutado no respeta la sintaxis. Para mas ayuda ejecutar el programa con -h o --help.

- Las próximas pruebas se refieren a errores en el archivo de entrada (Recordar que es un archivo de 80 caracteres, con todos 0 salvo el caracter central que es un 1):

  - ***$ ./autcel 30 15 inicial***
    Leyendo estado inicial...
    La cantidad de columnas en el archivo de entrada es mayor a la longitud indicada por "N".

  - ***$ ./autcel 30 90 inicial***
    Leyendo estado inicial...
    La cantidad de columnas en el archivo de entrada es menor a la longitud indicada por "N".

- Cambiando el archivo inicial de las maneras que muestra el comando *cat*:

  - ***$ cat inicial***
    hola
    $ ./autcel 30 80 inicial
    Leyendo estado inicial...
    El formato del archivo de entrada no es correcto.

  - ***$ cat inicial***
    0000000000000000000000000000000000000010000000000000000000000000000000000h
    $ ./autcel 30 80 inicial
    Leyendo estado inicial...
    El formato del archivo de entrada no es correcto.

2.5. Código

2.5.1. Código en C

proximo.h

```
#ifndef PROXIMO_H_
#define PROXIMO_H_

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

typedef enum { false, true } bool;

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


#endif /* PROXIMO_H_ */
```

proximo.c

```
#include "proximo.h"

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

  unsigned char izq,der,actual;
  if (j== 0)
    izq = *(a+N*i+ (N-1));
  else
    izq = *(a+N*i+ (j-1) );
  if (j == (N-1))
    der = *(a+N*i);
  else
    der = *(a+N*i+ (j+1) );
  actual = *(a+N*i+j);
  unsigned char pos = 4*izq + 2*actual + der;
  if ((regla) & (1<<(pos)))
    return 1;
  return 0;
}
```

main.c

```
#include "proximo.h"
#define TAMANIO_CELDA 4

bool verificarOpcion(char* argv1) {
  //Chequear que sea -V o --version, o bien, -h o --help
  if ( strcmp(argv1, "-V") == 0 || strcmp(argv1,"--version") == 0 )
    fprintf(stdout, "autcel: version 1.0\n");
  else if ( strcmp(argv1,"-h") == 0 || strcmp(argv1,"--help") == 0 )
    fprintf(stdout, "Uso:\nautcel -h\nautcel -V\nautcel R N inputfile [-
        o outputprefix]\n\nOpciones:\n-h, --help\tImprime este mensaje
        .\n-V, --version\tDa la version del programa.\n-o\t\tPrefijo de
        los archivos de salida.\n\nEjemplo:\nautcel 30 80 inicial -o
        evolucion\nCalcula la evolucion del automata \"Regla 30\", en un
        mundo unidimensional de 80 celdas, por 80 iteraciones.\nEl
        archivo de salida se llamara evolucion.pbm.\nSi no se da un
        prefijo para los archivos de salida, el prefijo sera el nombre del
        archivo de entrada.\n");
  else
    fprintf(stderr, "El comando ejecutado no respeta la sintaxis. Para
        mas ayuda ejecutar el programa con -h o --help.\n");
  return false;
}

int verificarRegla(char* argv1) {
  char* end;
  int regla = strtol(argv1, &end, 10);
  if (end == argv1 || *end != '\0' || errno == ERANGE) {
    fprintf(stderr, "El numero de regla no es valido. El mismo no
        contiene un valor numerico o se ha producido un overflow.\n");
    return -1;
  }
  if ((regla < 0) || (regla > 255)) {
    fprintf(stderr, "El numero de regla no es valido. Debe ser un valor
        numerico entre 0 y 255.\n");
    return -1;
  }
  return regla;
}

int verificarN(char* argv2) {
  char* end;
  int N = strtol(argv2, &end, 10);
  if (end == argv2 || *end != '\0' || errno == ERANGE) {
    fprintf(stderr, "El numero de celdas e iteraciones (\"N\") no es
        valido. El mismo no contiene un valor numerico o se ha
        producido un overflow.\n");
    return -1;
  }
```

```
  if ((N < 1) || (N > 10000)) {
    fprintf(stderr, "El numero de celdas e iteraciones (\"N\") no es
        valido. Debe ser un valor numerico entre 1 y 10000.\n");
    return -1;
  }
  return N;
}

FILE* verificarEntrada(char* argv3) {
  FILE* entrada = fopen(argv3, "r");
  if (entrada == NULL)
    fprintf(stderr, "El archivo de entrada especificado no se ha podido
        abrir o no existe.\n");
  return entrada;
}

void eliminarExtension(char* s) {
  char* c;
  c = strrchr (s, '.');
  if (c != NULL)
    *c = '\0';
}

bool verificarParametros(int argc, char* argv[], int* regla, int* N, FILE**
    entrada, char **salida) {
  //Recordar que argv tiene uno mas que los parametros que vienen de
      consola
  if (argc == 2)
    return verificarOpcion(argv[1]);
  else if (argc == 4 || argc == 6) {
    *regla = verificarRegla(argv[1]);
    if (*regla == -1) return false;

    *N = verificarN(argv[2]);
    if (*N == -1) return false;

    *entrada = verificarEntrada(argv[3]);
    if (*entrada == NULL) return false;

    if (argc == 6) {
      if ( strcmp(argv[4],"-o") != 0 ) {
        fprintf(stderr, "El comando ejecutado no respeta la sintaxis. Para
            mas ayuda ejecutar el programa con -h o --help.\n");
        return false;
      }
      *salida = argv[5];
      eliminarExtension(*salida);
    } else {
      *salida = argv[3];
      eliminarExtension(*salida);
```

```
    }
  } else {
    fprintf(stderr, "El comando ejecutado no respeta la sintaxis. Para
        mas ayuda ejecutar el programa con -h o --help.\n");
    return false;
  }
  return true;
}

unsigned char** crearMatriz(int N) {
  unsigned char** matriz;
  matriz = (unsigned char**) malloc(N * sizeof(unsigned char*));
  int i;
  for (i = 0; i < N; i++)
    matriz[i] = (unsigned char*) malloc(N * sizeof(unsigned char));
  return matriz;
}

void cargarPrimeraFila(int N, bool* continuar, FILE* entrada, unsigned
    char (*matriz)[N]) {
  int i = 0;
  int aux;
  while ((*continuar == true) && ((aux = fgetc(entrada)) != EOF)) {
    if (aux == '0' || aux == '1') {
      matriz[0][i] = aux - '0';
      i++;
    } else {
      if (aux != 10) {
        fprintf(stderr, "El formato del archivo de entrada no es correcto.\
            n");
        *continuar = false;
      } // else fprintf(stdout, "fin de linea\n");
    }
    if (i > N) {
      fprintf(stderr, "La cantidad de columnas en el archivo de entrada
          es mayor a la longitud indicada por \"N\".\n");
      *continuar = false;
    }
  }
  if ((*continuar == true) && (i < N)) {
    fprintf(stderr, "La cantidad de columnas en el archivo de entrada es
        menor a la longitud indicada por \"N\".\n");
    *continuar = false;
  }
}

void calcularFilas(bool continuar, int N, int regla, unsigned char(*matriz
    )[N]) {
  int i;
  int j;
```

```
  if (continuar == true) {
    for (i = 0; i < N - 1; i++) {
      for (j = 0; j < N; j++) {
        matriz[i + 1][j] = proximo(&matriz[0][0], i, j, regla, N);
      }
    }
  }
}

FILE* escribirImagen(bool continuar, int N, unsigned char (*matriz)[N],
    char** nombreSalida, bool* seAbrioArchSalida) {
  int i,j,k,l;
  FILE* salida;
  strcat(*nombreSalida, ".pbm");

  if ( (continuar == true) && ((salida = fopen(*nombreSalida, "wb")) ==
      NULL) ) {
    fprintf(stderr, "Error al crear archivo salida.\n");
    continuar = false;
  }
  if (continuar == true) {
    *seAbrioArchSalida = true;
        fprintf(salida, "P1\n%d %d\n", N*TAMANIO_CELDA, N*
            TAMANIO_CELDA);
    for (i = 0; i < N; i++) {
      for( k= 0; k < TAMANIO_CELDA; k++ ){
        for (j = 0; j < N; j++) {
          for( l= 0; l < TAMANIO_CELDA; l++ )
            fprintf(salida, "%d", matriz[i][j]);
        }
      fputc('\n', salida);
      }
    }
  }
  return salida;
}

void liberarMemoriaYCerrarArchivos(int N, unsigned char (*matriz)[N],
    FILE* entrada, FILE* salida, bool seAbrioArchSalida) {
  free(matriz);
  fclose(entrada);
  if (seAbrioArchSalida == true) fclose(salida);
}

int main(int argc, char *argv[]) {
  // En toda esta primera parte donde todavia no se pidio memoria dejo los
      return -1 que es mas comodo.
  int regla;
  int N;
  FILE* entrada;
```

```
  char* nombreSalida;

  if ( verificarParametros(argc, argv, &regla, &N, &entrada, &
       nombreSalida) == false ) return -1;

  bool continuar = true;

  unsigned char (*matriz)[N] = malloc(sizeof *matriz * N);
  if (!matriz)
    continuar = false;

  if (continuar == true) fprintf(stdout, "Leyendo estado inicial...\n");
  cargarPrimeraFila(N, &continuar, entrada, matriz);

  if (continuar == true) fprintf(stdout, "Calculando los %i estados
       siguientes...\n", N-1);
  calcularFilas(continuar, N, regla, matriz);

  if (continuar == true) fprintf(stdout, "Grabando %s.pbm\n",
       nombreSalida);
  bool seAbrioArchSalida = false;
  FILE* salida = escribirImagen(continuar, N, matriz, &nombreSalida, &
       seAbrioArchSalida);

  liberarMemoriaYCerrarArchivos(N, matriz, entrada, salida,
       seAbrioArchSalida);

  if (continuar == true) fprintf(stdout, "Listo.\n");
  return 0;
}
```

2.5.2. Código función proximo en Assembler

proximo.S

```
#include <mips/regdef.h>
#include <sys/syscall.h>

#Argumentos de la funcion
#define PROXIMO_ARG4  40
#define PROXIMO_ARG3  36
#define PROXIMO_ARG2  32
#define PROXIMO_ARG1  28
#define PROXIMO_ARG0  24

#Stack Size
#define PROXIMO_SS      24

#SRA
```

```
#define PROXIMO_FP      20
#define PROXIMO_GP      16

#LTA
#define POSICION        12
#define ACTUAL          8
#define DERECHA         4
#define IZQUIERDA       0

#NO HAY ABA NI RA YA QUE LA FUNCION PROXIMO ES UNA
    FUNCION HOJA.

.text
.align  2
.globl  proximo
.ent    proximo

#unsigned char proximo(unsigned char *a, unsigned int i, unsigned int j,
    unsigned char regla, unsigned int N)
proximo:
    #Instrucciones que van siempre
    .frame  $fp, PROXIMO_SS, ra
    .set    noreorder
    .cpload t9
    .set    reorder
    #Creo el stack frame
    subu    sp, sp, PROXIMO_SS
    .cprestore PROXIMO_GP
    sw      $fp,PROXIMO_FP(sp)
    sw      gp, PROXIMO_GP(sp)
    move    $fp,sp

#Guardo los parametros que me pasaron
#El parametro a4 ya esta en la pos 40 del stack -> 40($fp)
    sb      a3, PROXIMO_ARG3($fp) #Guardo a regla como byte
    sw      a2, PROXIMO_ARG2($fp) #J
    sw      a1, PROXIMO_ARG1($fp) #I
    sw      a0, PROXIMO_ARG0($fp) #A[0][0]

#Arranca la funcion con el primer if (j==0)
    lw      t0, PROXIMO_ARG2($fp) #Cargo j en t0
    bne     t0, zero, izqNoBorde    #Si j!=0 no es caso borde

izqBorde:
    lw      t0, PROXIMO_ARG4($fp) #Cargo N en t0
    lw      t1, PROXIMO_ARG1($fp) #Cargo I en t1
    mult    t0, t1                  #N*i
    mflo    t2                      #Guardo en t2 el resultado
    lw      t0, PROXIMO_ARG0($fp) #Cargo A[0][0] en t0
    addu    t3, t2, t0              #En t3 = N*i + A[0][0]
```

```
    lw      t0, PROXIMO_ARG4($fp) #Cargo N en t0
    addu    t3, t3, t0            #En t3 = t3(a[0][0] + N*i) + t0(N)
    addu    t4, t3, -1            #En t4 = t3 - 1
    lbu     t4, 0(t4)             #Cargo en t4 = *(t4)
    sb      t4, IZQUIERDA($fp)    #Guardo t4 (celda izquierda) en pos 0
        del stack
    b       derBorde              #Salto al calculo de la celda derecha

izqNoBorde:
    lw      t0, PROXIMO_ARG4($fp) #Cargo N en t0
    lw      t1, PROXIMO_ARG1($fp) #Cargo I en t1
    mult    t0, t1                #N*i
    mflo    t2                    #Guardo en t2 el resultado
    lw      t0, PROXIMO_ARG0($fp) #Cargo A[0][0] en t0
    addu    t3, t2, t0            #En t3 = N*i + A[0][0]
    lw      t0, PROXIMO_ARG2($fp) #Cargo J en t0
    addu    t3, t3, t0            #En t3 = t3(a[0][0] + N*i) + t0(J)
    addu    t4, t3, -1            #En t4 = t3 - 1
    lbu     t4, 0(t4)             #Cargo en t4 = *(t4)
    sb      t4, IZQUIERDA($fp)    #Guardo t4 (celda izquierda) en pos 0
        del stack

derBorde:
    lw      t0, PROXIMO_ARG4($fp) #Cargo N en t0
    addu    t0, t0, -1            #t0 = N - 1
    lw      t1, PROXIMO_ARG2($fp) #Cargo J en t1
    bne     t0, t1, derNoBorde    #Si j!= N-1 no es caso borde

    lw      t0, PROXIMO_ARG4($fp) #Cargo N en t0
    lw      t1, PROXIMO_ARG1($fp) #Cargo I en t1
    mult    t0, t1                #N*i
    mflo    t2                    #Guardo en t2 el resultado
    lw      t0, PROXIMO_ARG0($fp) #Cargo A[0][0] en t0
    addu    t3, t2, t0            #En t3 = N*i + A[0][0]
    lbu     t3, 0(t3)             #Cargo en t3 = *(t3)
    sb      t3, DERECHA($fp)      #Guardo t3 (celda derecha) en pos 4 del
        stack
    b       actual                #Salto para calcular el actual

derNoBorde:
    lw      t0, PROXIMO_ARG4($fp) #Cargo N en t0
    lw      t1, PROXIMO_ARG1($fp) #Cargo I en t1
    mult    t0, t1                #N*i
    mflo    t2                    #Guardo en t2 el resultado
    lw      t0, PROXIMO_ARG0($fp) #Cargo A[0][0] en t0
    addu    t3, t2, t0            #En t3 = N*i + A[0][0]
    lw      t0, PROXIMO_ARG2($fp) #Cargo J en t0
    addu    t3, t3, t0            #En t3 = t3(a[0][0] + N*i) + t0(J)
    addu    t4, t3, 1             #En t4 = t3 + 1
    lbu     t4, 0(t4)             #Cargo en t4 = *(t4)
```

```
    sb      t4, DERECHA($fp)      #Guardo t4 (celda derecha) en pos 4 del
        stack

actual:
    lw      t0, PROXIMO_ARG4($fp) #Cargo N en t0
    lw      t1, PROXIMO_ARG1($fp) #Cargo I en t1
    mult    t0, t1                #N*i
    mflo    t2                    #Guardo en t2 el resultado
    lw      t0, PROXIMO_ARG0($fp) #Cargo A[0][0] en t0
    addu    t3, t2, t0            #En t3 = N*i + A[0][0]
    lw      t0, PROXIMO_ARG2($fp) #Cargo J en t0
    addu    t3, t3, t0            #En t3 = t3(a[0][0] + N*i) + t0(J)
    lbu     t3, 0(t3)             #Cargo en t3 = *(t3)
    sb      t3, ACTUAL($fp)       #Guardo t3 (celda actual) en pos 8 del
        stack

posicion:
    lbu     t0, IZQUIERDA($fp)    #Cargo celda izquierda en t0
    lbu     t1, DERECHA($fp)      #Cargo celda derecha en t1
    lbu     t2, ACTUAL($fp)       #Cargo celda actual en t2
    sll     t3, t0, 2             #En t3 = izquierda * 4
    sll     t4, t2, 1             #En t4 = actual * 2
    addu    t5, t3, t4            #En t5 = t3(4izq) + t4(2*act)
    addu    t5, t5, t1            #En t5 = t5 + t1(der)
    sb      t5, POSICION($fp)     #Guardo t5 (pos) en pos 12 del stack

comparacion:
    lbu     t0, PROXIMO_ARG3($fp) #Cargo Regla en t0
    lbu     t1, POSICION($fp)     #Cargo Pos en t1
    li      t2, 1                 #Cargo 1 en t2
    sll     t3, t2, t1            #En t3 = (1<<(pos))
    and     t4, t0, t3            #En t4 = (regla & t3)
    beq     t4, zero, devuelvo0   #Si t4 = 0, return 0
    li      v0, 1                 #Sino return 1
    b       salir                 #Salgo de la funcion

devuelvo0:
    move    v0, zero              #Return 0

salir :
#Destruyo el stack frame
    move    sp, $fp
    lw      $fp,PROXIMO_FP(sp)
    lw      gp, PROXIMO_GP(sp)
    addu    sp, sp,PROXIMO_SS
    j       ra

.end    proximo
```

2.5.3. Código del Makefile

Makefile

```
CC = gcc
CFLAGS = -g -c
PROG = autcel

c_proximo: src/proximo.c
	$(CC) $(CFLAGS) src/proximo.c

as_proximo: src/proximo.S
	$(CC) $(CFLAGS) src/proximo.S

tp1_orga: src/main.c
	$(CC) $(CFLAGS) src/main.c

mips: as_proximo tp1_orga
	$(CC) proximo.o src/main.c -o $(PROG)

c: c_proximo tp1_orga
	$(CC) proximo.o src/main.c -o $(PROG)

clean:
	rm -rf *.o $(PROG)
```

2.6. Diagrama de Stack

2.6.1. Función proximo

Aquí presentamos un diagrama del Stack de la función *proximo*. Vale aclarar que en el mismo, se incluye el ABA (Argument Building Area) de la función caller (En este caso la función *calcularFilas*). Además, como la cantidad de argumentos que recibe *proximo* es mayor que 4, necesitamos agregarlos a "mano" y, para que quede múltiplo de 8 bytes, agregamos un padding. A su vez, al ser *proximo* una función "hoja", su stack no posee ABA.

Por otra parte también consideramos importante mencionar que tanto el SRA (Saved Register Area, que incluye al GP y FP) como LTA (Local Temporary Area, que incluye a las varibles utilizadas en la función) son múltiplo de 8 bytes, por lo que no requieren del agregado de un padding.

Stack Size función *proximo*: 24

PADDING	44
A4	40
A3	36
A2	32
A1	28
A0	24
FP	20
GP	16
POS	12
ACTUAL	8
DER	4
IZQ	0

Cuadro 1: Stack de la función proximo

3. Resultados

En las figuras 1, 2 y 3 podemos observar los resultados de las corridas pedidas por los docentes de la materia. Todas ellas parten de un archivo inicial de 80 caracteres con una celda ocupada en el centro:
$ cat inicial
00000000000000000000000000000000000000000010000000000000000000000000000000000000000000

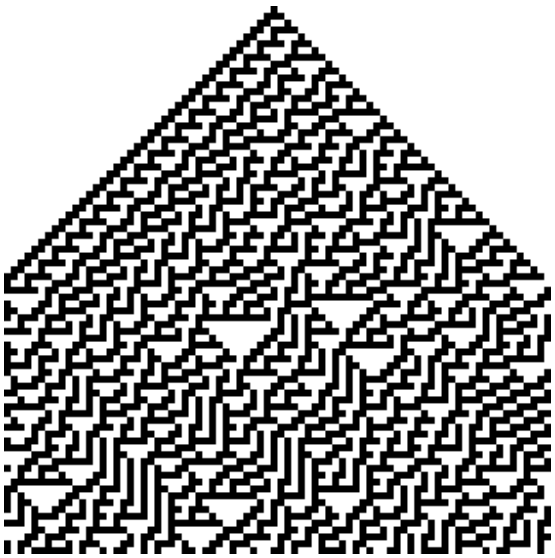

Figura 1: Regla 30.

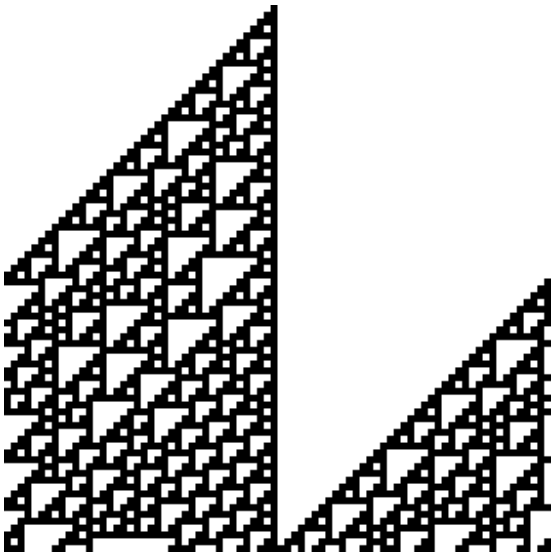

Figura 2: Regla 110.

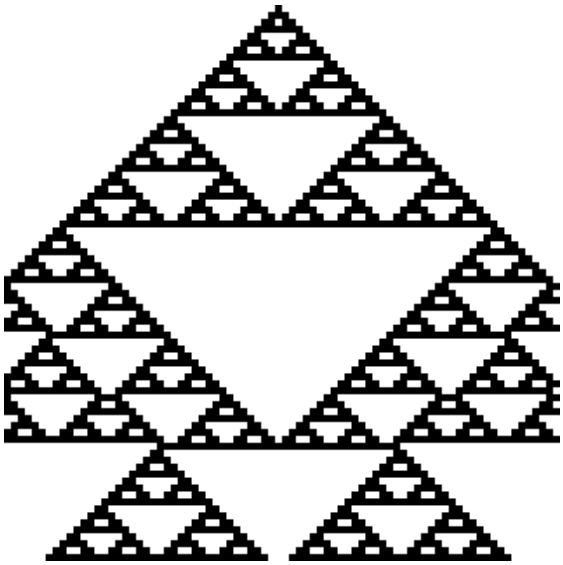

Figura 3: Regla 126.

Podemos observar que en los casos de las reglas 30 y 110 se corresponde con el resultado esperado [6][7]; no así en el caso de la regla 126, donde se puede ver que lo esperado [8] no se corresponde con la figura 3. La explicación que encontramos es que el resultado mostrado en [8] posee 80 iteraciones (alto), pero de ancho posee más pixels. Debido a esto y a que se utiliza la hipótesis del mundo toroidal[1], en el momento en que los pixels negros llegan a los extremos, los vecinos de una y otra "versión" pueden diferir y de aquí es que surgen las diferencias. En la figura 4 podemos ver una versión de la regla 126 con un N mayor, por lo que hasta la iteración 80 (y más también) se está en "igualdad de condiciones" con nuestra referencia. Aquí sí podemos observar que ambas imágenes son iguales.

[1]Hipótesis del mundo toroidal: la columna $N-1$ pasa a ser vecina de la columna 0, por lo que el vecino izquierdo de la celda [i, 0] es el [i, $N-1$], y viceversa. Recordarla también a la hora de hacer la comparación de imágenes.

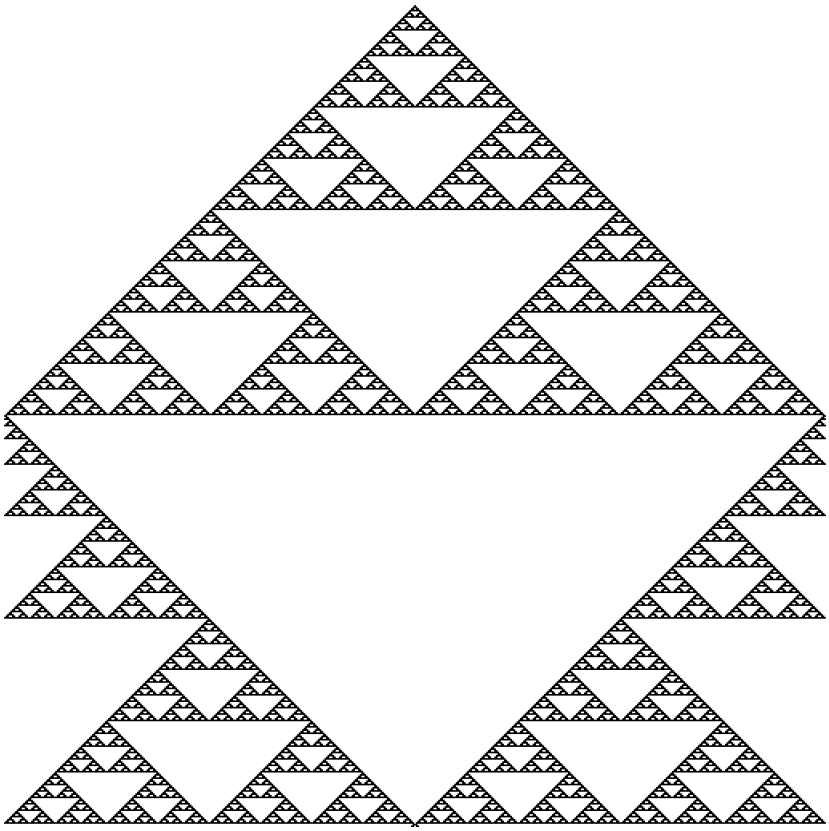

Figura 4: Regla 126 con N=512.

4. Conclusiones

De este trabajo práctico se pudo aprender cómo programar con el conjunto de instrucciones Assembly de MIPS32, así como también la utilización correcta de la ABI propuesta por la cátedra, entendiendo de esta manera cómo funciona una computadora a bajo nivel.

El crear la función proximo tanto en C como en Assembler nos permitió entender como las instrucciones de un lenguaje de mayor nivel pueden ser traducidas en un conjunto de instrucciones de bajo nivel. Esto también nos posibilitó observar las diferencias entre el código de Assembler equivalente al de C escrito por nosotros comparado con el generado con *gcc*, notando que este último es optimizado por el compilador y por lo tanto tendrá una mejor performance.

5. Referencias

[1] The NetBSD project, http://www.netbsd.org/.
[2] GXemul, http://gavare.se/gxemul/.
[3] Autómatas celulares elementales: http://mathworld.wolfram.com/ElementaryCellularAutomaton.html.
[4] func_call_conv.pdf, en el área de Material de los archivos del grupo Yahoo del curso.
[5] Formato PBM: http://netpbm.sourceforge.net/doc/pbm.html.
[6] Regla 30: http://www.wolframalpha.com/input/?i=rule+30%2C+80+steps
[7] Regla 110: http://www.wolframalpha.com/input/?i=rule+110%2C+80+steps
[8] Regla 126: http://www.wolframalpha.com/input/?i=rule+126%2C+80+steps

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 GXemul [1] para simular el entorno de desarrollo de una máquina MIPS corriendo una versión reciente del sistema operativo NetBSD [3]. GXemul se puede hacer correr bajo Windows, en el entorno Cygwin [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 [5], 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 estar ocupadas 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 ocupado 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 [8] como se ve en la Figura 2.

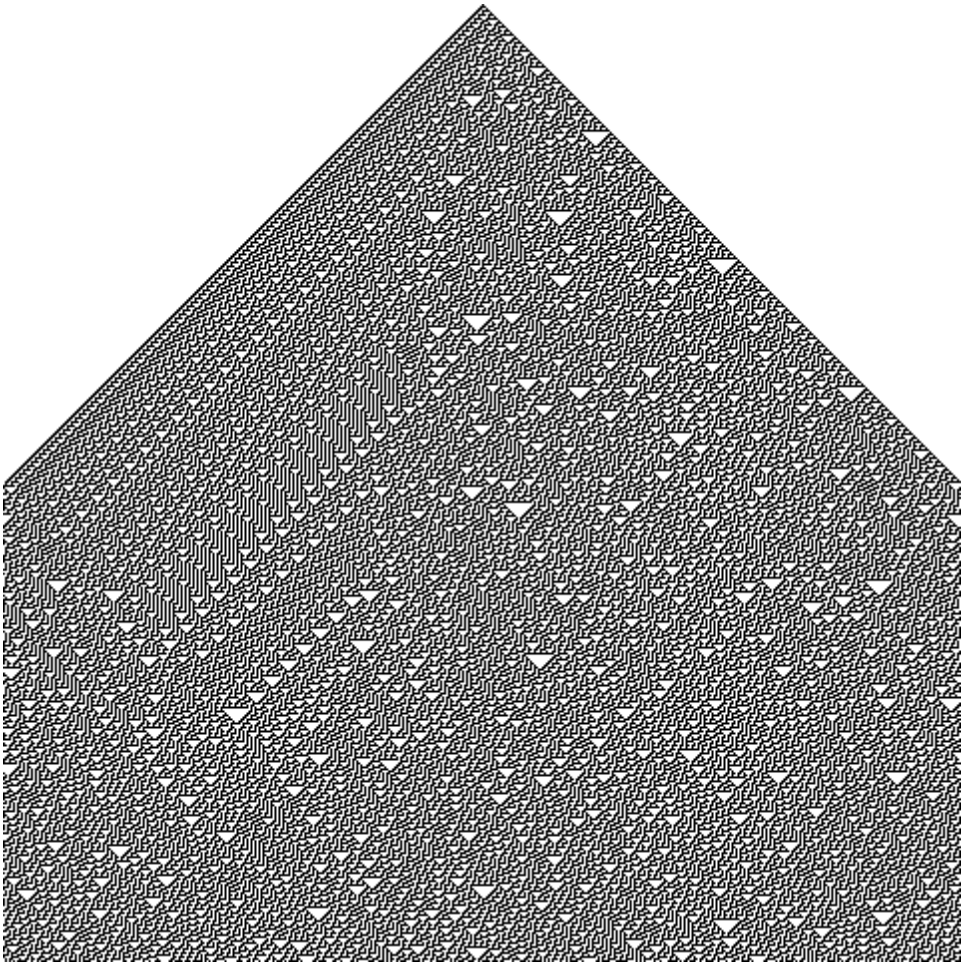

Figura 1: Regla 30 a partir de una celda ocupada en el centro

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 [6] 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`.

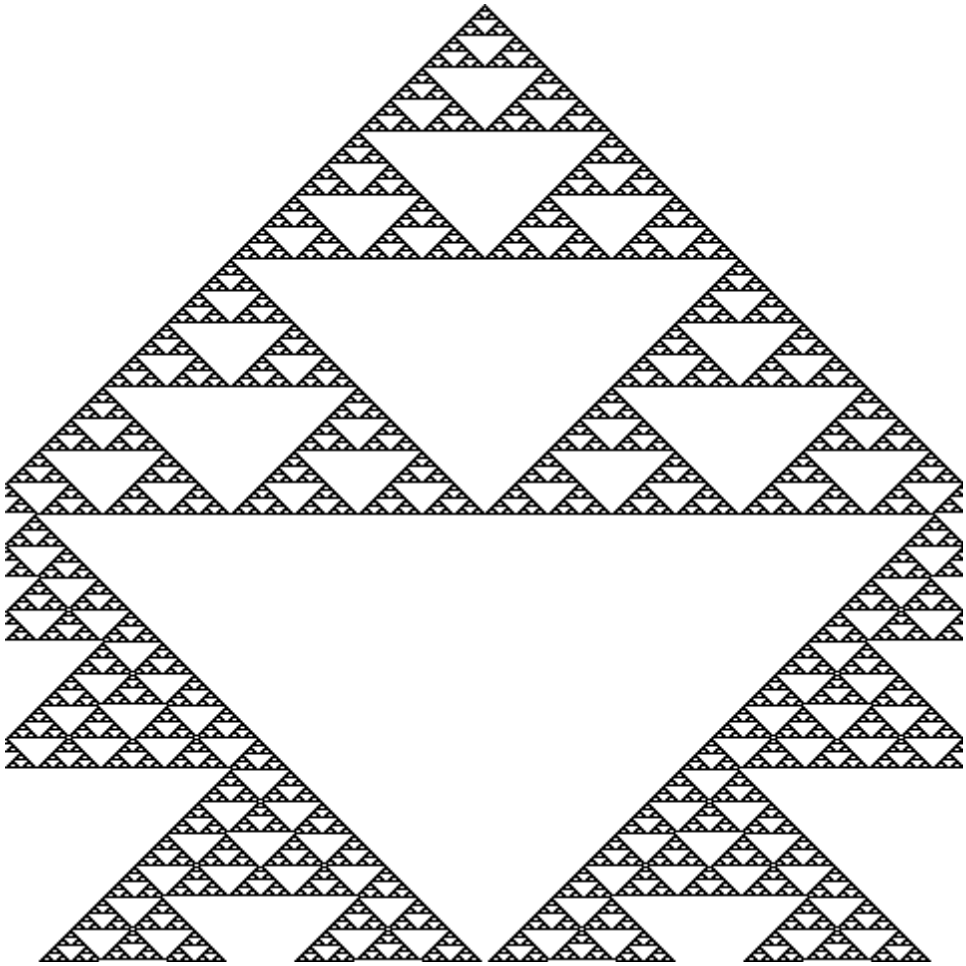

Figura 2: Regla 126 a partir de una celda ocupada en el centro

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, como se ve en la Figura 3. En este caso, por supuesto, nuestro mundo es en realidad unidimensional y los vecinos superiores e inferiores de una celda no nos interesan.

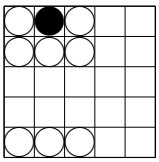

Figura 3: Ejemplo de mundo toroidal: la celda (0,1) y sus vecinos.

6.2. Comportamiento deseado

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

```
$ 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, NetBSD/pmax.

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 [6] 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" [4].

7.4. Algoritmo

El algoritmo a implementar es el algoritmo de autómatas celulares unidimensionales[7], 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 NetBSD 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 dos formatos: digitalizado e impreso en papel.

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 24 de Septiembre de 2015.[3] Devolución: Jueves 1 de Octubre de 2015. La última fecha de entrega y presentación es el jueves 8 de Octubre de 2015.

Referencias

[1] GXemul, `http://gavare.se/gxemul/`.

[2] Installing the MIPS Environment over Cygwin on Windows, `http://faculty.cs.tamu.edu/bettati/Courses/410/2006B/Projects/gxemulcygwin.html`

[3] The NetBSD project, `http://www.netbsd.org/`.

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

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

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

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

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

[3] La RAE dice que los nombres de los meses van en minúscula. También dice que Septiembre va sin p. No es lo que era, la RAE.