Asignatura:
66.20 Organización de computadoras

Profesor: Hamkalo, José Luis

Trabajo Práctico:
Conjunto de instrucciones MIPS

Nombre y apellido	Padrón	Correo electrónico	Slack
Julian Mejliker	100866	jmejlker@fi.uba.ar	Julian Mejliker
Tomas Nocceti	100853	nocetti.tomas@gmail.com	Tomas Nocetti
Netanel Jamilis	99093	njamilis@fi.uba.ar	Netanel Jamilis

Primer cuatrimestre 2020

Índice

1. Introducción

Una vez instalado el entorno de trabajo de assembly MIPS32 junto con las herramientas necesarias se comenzó con el desarrollo del trabajo práctico numero 1, el cual consiste en la implementacion del juego de la vida en el lenguaje en C, y tambien la implementacion de una funcion especifica en el leguaje de Assembly MIPS.

La implementacion del juego, fue realizada por los integrantes del grupo siguiendo las directivas del mismo. El juego consiste en una grilla de NxM celdas, la cuales poseen dos estados, vivas o muertas. En cada iteracion del juego, se evalua para cada celda su estado sigueinte, dependiendo de la cantidad de vecinos vivos o muertos, calculado por la funcion Vecinos. Para la implementacion de la funcion en cuestion se tomo como base la funcion desarrollada en C, y se realizo una traduccion, sin ayuda del compilador.

2. Diseño e implementación

2.1. Linea de comando

Para la corrida del programa se deben seguir esta opciones.

```
./tpi_cells <iterations> <width> <height> <filename> -o <prefix>
 -h --help: Print this message and exit
 -v --version: Da la version del programa
 -o : Prefijo de los archivos
```

Se implemento un sistema de validaciones para que el uso del programa sea amigable y se pueda desarrollar sin problemas.

2.2. Desarrollo del Codigo Fuente

A partir de comprender correctamente los lineamientos del juego y las indicaciones por parte de la practica, se procedio al desarrollo del programa en el lenguaje C. Para simular la grilla de NxM se implemeto una struct ***cellsgrid*** el cual posee la info de la grilla y un puntero a un array de NxM celdas, la cual son 1 y 0. Luego se prosiguio al desarrollo de todas las funcionalidades. Una vez terminado el desarrollo, se prosiguio a traducir la funcion ***vecinos***.

Utilizando las indicaciones dadas en clase y las normas a la hora de crear el ABI, se utilizo todas las practicas para el optimo desarrollo de la funcion.

3. Proceso de Compilacion

Para la correcta compilacion del programa, se definio dos distintos objs en el archivo Makefile. Para generar cada uno de los ejecutables se debe correr distintas lineas:

- make tp1_cells: este comando genera el ejecutable sin el codigo assembler
- make tp1_cells_asm: este comando genera el ejecutable con el codigo assembler(Solo soportado en ambiente MIPS)

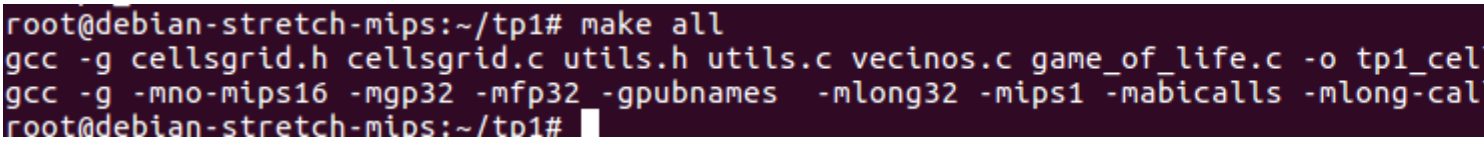

Figura 1: Corridas

4. Casos de prueba

```
root@debian-stretch-mips:~/tp1# ./tp1_cells_asm 10 20 20 sapo -o sapo-estado
Leyendo estado inicial..
Grabando sapo-estado_0.pbm
Grabando sapo-estado_1.pbm
Grabando sapo-estado_2.pbm
Grabando sapo-estado_3.pbm
Grabando sapo-estado_4.pbm
Grabando sapo-estado_5.pbm
Grabando sapo-estado_6.pbm
Grabando sapo-estado_7.pbm
Grabando sapo-estado_8.pbm
Grabando sapo-estado_9.pbm
Grabando sapo-estado_10.pbm
Listo
root@debian-stretch-mips:~/tp1# ./tp1_cells_asm 10 20 20 pento -o pento-estac
Leyendo estado inicial..
Grabando pento-estado_0.pbm
Grabando pento-estado_1.pbm
Grabando pento-estado_2.pbm
Grabando pento-estado_3.pbm
Grabando pento-estado_4.pbm
Grabando pento-estado_5.pbm
Grabando pento-estado_6.pbm
Grabando pento-estado_7.pbm
Grabando pento-estado_8.pbm
Grabando pento-estado_9.pbm
Grabando pento-estado_10.pbm
Listo
root@debian-stretch-mips:~/tp1# ./tp1_cells_asm 10 20 20 glider -o glider-est
Leyendo estado inicial..:~/tp1# ./tp1_cells_asm 10 20 20 glider -o glider-est
Grabando glider-estado_0.pbm
Grabando glider-estado_1.pbm
Grabando glider-estado_2.pbm
Grabando glider-estado_3.pbm
Grabando glider-estado_4.pbm
Grabando glider-estado_5.pbm
Grabando glider-estado_6.pbm
Grabando glider-estado_7.pbm
Grabando glider-estado_8.pbm
Grabando glider-estado_9.pbm
Grabando glider-estado_10.pbm
Listo
```

Figura 2: Corridas

5. Vista general del stack

Aqui se muestra un diagrama de como es la estructura del stack en la funcion *vecinos* en assembler.

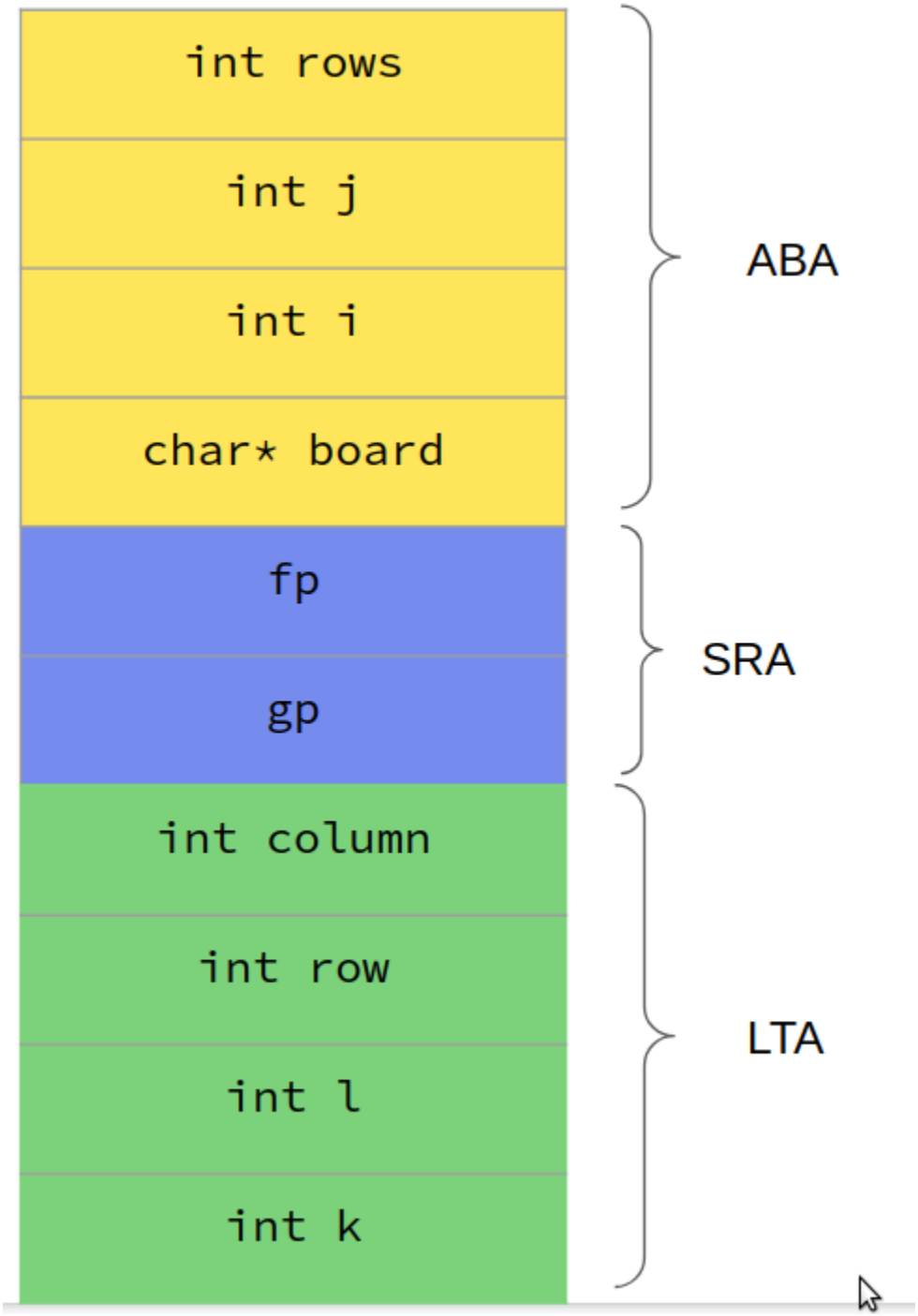

Figura 3: Corridas

6. Conclusión

Se corrio en el ambiente del emulador de mips, los dos programas, es decir uno con codigo asm y el otro no. Vamos a comparar los tiempos de distintas corridas.

```
root@debian-stretch-mips:~/tp1# time ./tp1_cells_asm 10 400 400 glider -o salida
Leyendo estado inicial..:
Grabando salida_0.pbm
Grabando salida_1.pbm
Grabando salida_2.pbm
Grabando salida_3.pbm
Grabando salida_4.pbm
Grabando salida_5.pbm
Grabando salida_6.pbm
Grabando salida_7.pbm
Grabando salida_8.pbm
Grabando salida_9.pbm
Grabando salida_10.pbm
Listo

real 2m38.615s
user 2m36.036s
sys 0m2.112s

root@debian-stretch-mips:~/tp1# time ./tp1_cells 10 400 400 glider -o salida
Leyendo estado inicial..
Grabando salida_0.pbm
Grabando salida_1.pbm
Grabando salida_2.pbm
Grabando salida_3.pbm
Grabando salida_4.pbm
Grabando salida_5.pbm
Grabando salida_6.pbm
Grabando salida_7.pbm
Grabando salida_8.pbm
Grabando salida_9.pbm
Grabando salida_10.pbm
Listo

real 2m35.888s
user 2m32.504s
sys 0m2.388s

root@debian-stretch-mips:~/tp1# time ./tp1_cells_asm 10 10 10 glider -o salida
Leyendo estado inicial..
Grabando salida_0.pbm
Grabando salida_1.pbm
Grabando salida_2.pbm
```

```
Grabando salida_3.pbm
Grabando salida_4.pbm
Grabando salida_5.pbm
Grabando salida_6.pbm
Grabando salida_7.pbm
Grabando salida_8.pbm
Grabando salida_9.pbm
Grabando salida_10.pbm
Listo

real 0m0.186s
user 0m0.148s
sys 0m0.036s
root@debian-stretch-mips:~/tp1# time ./tp1_cells 10 10 10 glider -o salida
Leyendo estado inicial..
Grabando salida_0.pbm
Grabando salida_1.pbm
Grabando salida_2.pbm
Grabando salida_3.pbm
Grabando salida_4.pbm
Grabando salida_5.pbm
Grabando salida_6.pbm
Grabando salida_7.pbm
Grabando salida_8.pbm
Grabando salida_9.pbm
Grabando salida_10.pbm
Listo

real 0m0.152s
user 0m0.140s
sys 0m0.008s
```

Por lo que se puede ver las corridas en asm, no son mas rapidas que el programa todo en C como era de esperar. A partir de esto se puede deducir, que no siempre utilizar ASM es mejor que utilizar codigo C u otro de mas alto nivel. Por otro lado, tambien se puede deducir que no realizamos las mejores optimizaciones en codigo ASM, debido al poco conocimiento del mismo. Entonces hay que estudiar bien en cada situacion si la implementacion de parte del codigo es ASM es una optimizacion o no.

7. Anexo

7.1. vecinos.S

```
.globl vecinos

#define PADDING_X 1
#define PADDING_Y 1

# Stack size
#define SS 24

# Offset to each argument in the ABA
#define O_ARG0(sp) (SS + 0)(sp)
#define O_ARG1(sp) (SS + 4)(sp)
#define O_ARG2(sp) (SS + 8)(sp)
#define O_ARG3(sp) (SS + 12)(sp)
#define O_ARG4(sp) (SS + 16)(sp)

#define O_LTA_K(sp) 0(sp)
#define O_LTA_L(sp) 4(sp)
#define O_LTA_ROW(sp) 8(sp)
#define O_LTA_COLUMN(sp) 12(sp)

# Offset to fp and gp
#define O_FP(sp) (SS-4)(sp)
#define O_GP(sp) (SS-8)(sp)

# ##############################################################################
#
# int vecinos(unsigned char* board, unsigned int i, unsigned int j, unsigned int rows, unsi
#
# Assembly implementation of the vecinos function.
#
# Stack usage:
#  [SS - O_GP]
#  [SS - O_FP]
#
# Register usage:
#  -- Arguments --
# $a0: char* board
# $a1: unsigned int i
# $a2: unsigned int j
# $a3: unsigned int rows
# $t7: unsigned int columns
#
# $t0: i iterator for outter
```

```
# $t1: i iterator for inner
# ##############################################################################
vecinos:
    addiu   $sp, $sp, -SS             # Create the stack frame

    # Set up ABA and load fifth argument into registers
    sw      $a0, O_ARG0($sp)         # Store $a0 in the ABA
    sw      $a1, O_ARG1($sp)         # Store $a1 in the ABA
    sw      $a2, O_ARG2($sp)         # Store $a2 in the ABA
    sw      $a3, O_ARG3($sp)         # Store $a3 in the ABA
    lw      $t7, O_ARG4($sp)         # Load fifth argument (iterations)

    # Save LTA
    li      $t0, -1
    sw      $t0, O_LTA_K($sp)

    li      $t1, -1
    sw      $t1, O_LTA_L($sp)

    sw      $zero, O_LTA_ROW($sp)
    sw      $zero, O_LTA_COLUMN($sp)

    # SRA
    sw $fp, O_FP($sp) # Store fp in the ABA
    sw $gp, O_GP($sp) # Store gp in the ABA

    # Set neighbout count
    addi  $v0, $zero, 0

    # Start for loop
  outer_loop:
    bge   $t0, 2, end

    # Set row neightbour position
    add   $t2, $a1, $t0
    add   $t2, $t2, $a3
    remu  $t2, $t2, $a3
    sw    $t2, O_LTA_ROW($sp)

    #fijo la proxima posicion
    addi $t0, $t0, 1
    sw   $t0, O_LTA_K($sp)
  inner_loop:
    bge  $t1, 2, outer_loop

    # Set column neightbour position
    add   $t3, $a2, $t1
```

```
  add   $t3, $t3, $t7
  remu  $t3, $t3, $t7
  sw    $t3, O_LTA_COLUMN($sp)

  # Find and add neightbour
  mul   $t4, $t3, $a3 # multiply $t4 = (row) * (column-neighbour)
  add   $t4, $t4, $t2 # add $t4 = $t4 + (row-neighbour)
  add   $t4, $t4, $a0 # add address and save to t4.
  lb    $t5, 0($t4)
  add   $v0, $v0, $t5

  # Check not to sum original position
  bne   $t0, 1, add2
  addi  $t1, $t1, 1
add2:
  addi  $t1, $t1, 1
  sw    $t1, O_LTA_L($sp)
  b     inner_loop

end:
  # Stack unwind
  lw    $fp, O_FP($sp)
  lw    $gp, O_GP($sp)
  addiu $sp, $sp, SS

  # Return
  jr  $ra
```