CSC231 - Assembly

Week #10

Dominique Thiébaut
dthiebaut@smith.edu
Exploring Shared Structures in GPS Trajectory Data Under Uncertainty

Dr. Yang Li, '11
Talk today
12:10 p.m.
FH241
Problem of the Day(s):
Implement 1D Game of Life in Assembly!
How to Approach This?
#Step 1: Write Algorithm in an More Comfortable Language...
# gameOfLife.py
# D. Thiebaut
# 1-Dimensional Game of Life

from __future__ import print_function
from __future__ import division
import random

def life( dish, N ):
    newGen = ""
    for i in range( 0, N ):
        neighbors = 0
        if i>0 and dish[i-1]!=' ': neighbors += 1
        if i < N-1 and dish[i+1]!=' ': neighbors += 1
        if neighbors == 1:
            newGen += "#"
        else:
            newGen += " "
    return newGen

def main():
    N = 40
    dish = (N//2-10)*"#" + 10*" #" + (N//2-10)*" 
    dish = dish[0:N]

    # print first generation
    print( dish )

    # repeat, for some generations
    for generation in range( 20 ):
        newGen = life( dish, N )
        print( newGen )
        dish = newGen

main()}
# gameOfLife.py
# D. Thiebaut
# 1-Dimensional Game of Life

from __future__ import print_function
from __future__ import division
import time

def life( dish, N):
    newGen = [0] * N
    for i in range(1, N-1):
        fate = dish[i-1] ^ dish[i+1] ^ is xor in Python!
    newGen[i] = fate
    return newGen

def printDish( dish):
    cells = [' ', '#']
    for c in dish:
        print( cells[c], end="")
    print()

def main():
    N = 40
    dish = (N//2-10)[1] + 10*[0,1] + (N//2-10)[0]
    dish = dish[0:N]
    printDish( dish)

    for generation in range(20):
        newGen = life( dish, N )
        printDish( newGen )
        dish = newGen
        time.sleep( 1 )

main()
Develop Assembly Program as a Class Exercise
```asm
;;; gameOfLife.asm
;;; D. Thiebaut
;;; 1-Dimensional Game of Life

N = 140

;;; # start with some pattern of cells
;;; dish = (N/2) * " " + 1 * " #" + (N/2) * " 
;;; dish = dish[0:N] # truncate to be safe

;;; # print first generation
;;; print( dish )

;;; # repeat for some number of generations

for generation in range( 60 ):
    newGen ="

    # loop through all the cells of dish
    for i in range( 0, N):
        neighbors = 0
        if i >0 and dish[i-1] != ' ': neighbors += 1
        if i <N-1 and dish[i+1] != ' ': neighbors += 1

        # add a live or dead cell to the new generation
        if neighbors == 1:
            newGen += "#"
        else:
            newGen += " "

    print( newGen )
    dish = newGen
```

---

Step 1
D. Thiebaut, Computer Science, Smith College
section .text
extern _printString
extern _println
extern _printInt
global _start

_start:

; print( dish )

    mov    ecx, N
    mov    esi, dish
    ; print N characters

for1:
    mov    dword[saveEcx1], ecx
    ; esi points to array to print
    ; save ecx, temporarily

    mov    eax, 0
    mov    al, byte[esi]
    ; eax now contains 0 or 1
    mov    ecx, eax
    ; put eax into ecx

    add    ecx, cells
    ; make ecx point to ' ' or '#'

    mov    edx, 1
    call   _printString
    ; get ready to print 1 char
    ; print string of 1 char, either ' ' or '#'

    mov    ecx, dword[saveEcx1]
    ; restore ecx

    add    esi, 1
    ; esi points to next cell of array

    loop   for1
    ; keep looping

    call   _println
    ; go to next line when array printed
for generation in range(...):

; # repeat for some number of generations
; for generation in range(60):
    mov ecx, 60
for2:    mov dword[saveEcx2], ecx
         ; save num gen in saveEcx

; ;; return to for generation loop
    mov ecx, dword[saveEcx2]
    loop toFor2

; ;; exit
    mov eax, 1
    mov ebx, 0
    int 0x80

; ;; relay jump
toFor2:    jmp for2
(END)
Compute Fate of NewGen
dish = newGen

;   dish = newGen
    mov   ecx, N
    mov   esi, newGen
    mov   edi, dish
for5:  mov   al, byte[esi]
    mov   byte[edi], al
    inc   esi
    inc   edi
    loop  for5
A Look at the Assembly Code...
Serpinsky Triangle

https://www.youtube.com/watch?v=wcxmdiuYjhk
If-statements in Assembly
• **Jmp**: the jump instruction

• **flags** register

• **conditional** jumps (jne, je, jgt, jge, jlt, jle, ja, jb...)

Jumping around…

Start:

```assembly
mov     ebx, Table          ;
jmp     there               ;

here:   mov     al, 1           ;
        mov     ecx, N          ;

there:  mov     byte[ebx+esi], al ;
        inc     esi            ;
        add     al, al          ;
jmp     here                ;
```
Jumping around...

_Start:

mov ebx, Table ;

jmp there ;

here:

mov al, 1 ;
mov ecx, N ;

there:

mov byte[ebx+esi], al ;
inc esi ;
add al, al ;
jmp here ;
Jumping around...

```assembly
_Start:
    mov     ebx, Table ;
    jmp _there ;

here:   mov     al, 1 ;
        mov     ecx, N ;

_there: mov     byte[ebx+esi], al ;
        inc     esi ;
        add     al, al ;
    jmp     here ;
```
_Start:
  mov  ebx, Table ;
  jmp  there ;

here:  mov  al, 1 ;
       mov  ecx, N ;

there: mov  byte[ebx+esi], al ;
       inc  esi ;
       add  al, al ;
  jmp  here ;
Jumping around…

_Start:

```
mov    ebx, Table          ;
jmp   there               ;
here:  mov     al, 1        ;
mov    ecx, N             ;
there: mov     byte[ebx+esi], al ;
    inc     esi            ;
    add     al, al         ;
jmp    here              ;
```
Jumping around...

_Start:
  mov     ebx, Table          ;
  jmp     there               ;
_here:   mov     al, 1               ;
          mov     ecx, N              ;
_there:  mov     byte[ebx+esi], al   ;
          inc     esi                 ;
          add     al, al              ;
          jmp     here               ;
Jumping around…

_Label:_

```assembly
_start:
    mov    ebx, Table          ;
    jmp    there               ;

here:   mov    al, 1           ;
        mov    ecx, N          ;

there:  mov    byte[ebx+esi], al ;
        inc    esi           ;
        add    al, al        ;
        jmp    here          ;
```
Jumping around…

_Start:
    mov    ebx, Table          ;
    jmp    there
    ;

_here:  mov    al, 1               ;
        mov    ecx, N              ;

_there: mov    byte[ebx+esi], al ;
        inc    esi                ;
        add    al, al             ;
    jmp    here                ;
jmp there ; "mov eip, there"
Flags Register
eax
ebx
cyx
dex
esi
edi

CF PF ZF SF OF DF

ALU

ADD
AND
### Examples

<table>
<thead>
<tr>
<th>Instruction</th>
<th>AL Value</th>
<th>Immediate Value</th>
<th>CF PF AF SF IF ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov al, 0x43</td>
<td>67</td>
<td>0x43</td>
<td></td>
</tr>
<tr>
<td>sub al, 0x43</td>
<td>66</td>
<td>0x42</td>
<td>IF ID</td>
</tr>
<tr>
<td>mov al, 0x44</td>
<td>68</td>
<td>0x44</td>
<td>CF PF AF SF IF ID</td>
</tr>
<tr>
<td>sub al, 0xff</td>
<td>255 or -1</td>
<td>0xff</td>
<td>CF PF AF IF ID</td>
</tr>
<tr>
<td>mov al, 0x81</td>
<td>67</td>
<td>0x81</td>
<td>129 or -127 CF SF IF OF ID</td>
</tr>
</tbody>
</table>
Conditional Jumps
Example with Jnz

_Start:
    mov ecx, 10
_for:  ...

dec ecx ; ecx ← ecx - 1
_jnz for ; if previous op didn't result in 0 in ALU
    ; then jump
    ; else continue here...
We stopped here last time...
Conditional Jump

```assembly
xxx  op1, op2 ; ALU operation
      ; carry bit possibly
jc   there   ; set
yyy  op1, op2 ; jump if carry to
...               ;
there: zzz  op1, op2 ; otherwise do this
```
```
xxx   op1, op2 ; ALU operation
       ; carry bit possibly set
jc     there    ; jump if carry to
yyy   op1, op2 ; otherwise do this
...
there: zzz   op1, op2
```

Case 1: carry gets set by xxx
Conditional Jump

Case 2: carry not set by \( \text{xxx} \)
Family of Conditional Jumps

- JE, JZ
- JNE, JNZ
- JG, JGE, JNL
- JL, JLE, JNG
Family of Conditional Jumps

EAX: 0xFFFF FFFF
EBX: 0x0000 0001

Which is greater?

• JE, JZ
• JNE, JNZ
• JG, JGE, JNL
• JL, JLE, JNG
Family of Conditional Jumps

EAX: 0xFFFF FFFF  
EBX: 0x0000 0001  \{  Which is greater?

- JE, JZ
- JNE, JNZ
- JG, JGE, JNL
- JL, JLE, JNG
- JE, JZ
- JNE, JNZ
- JA, JAE, JNB
- JB, JBE, JNA
How do we compare two quantities?
; if (a==b)
;     c = 3
; else
;     c = -1

    mov   eax, dword[a]        ;eax ← a
    sub   eax, dword[b]        ;eax ← a - b, set flags
    jnz   else                ;if ZF flag not set, go to else

then:   mov   dword[c], 3    ;otherwise, a==b, set c to 3
    jmp   done                 ;and skip else part

else:   mov   dword[c], -1    ;a!=b, set c to -1

done:   .  .  .
; if (a==b)
;     c = 3
; else
;     c = -1

mov     eax, dword[a] ; eax <- a
sub     eax, dword[b] ; eax <- a - b, set flags
jnz     else ; if ZF flag not set, go to else

then:   mov     dword[c], 3 ; otherwise, a==b, set c to 3
        jmp     done ; and skip else part

else:   mov     dword[c], -1 ; a!=b, set c to -1

done:    .  .  .
sub is ok, but it modifies the dest operand

; if (a==b)
;     c = 3
; else
;     c = -1

mov     eax, dword[a]        ;eax ← a
sub     eax, dword[b]        ;eax ← a - b, set flags
jne     else                  ;if ZF flag not set, go to else

then:   mov     dword[c], 3      ;otherwise, a==b, set c to 3
        jmp     done            ;and skip else part

else:    mov     dword[c], -1     ;a!=b, set c to -1

done:    . . .
cmp is better, it subtracts b from eax, but does not modify eax

; if (a==b)
;     c = 3
; else
;     c = -1

mov     eax, dword[a]          ;eax <- a
cmp     eax, dword[b]          ;compute a - b, set flags
jne     else                    ;if ZF flag not set, go to else

then:   mov     dword[c], 3      ;otherwise, a==b, set c to 3
        jmp     done                ;and skip else part

else:   mov     dword[c], -1     ;a!=b, set c to -1

done:    . . .
Another example with cmp

```plaintext
; int a, c    // signed ints
; if (a < 10)
    c = 3
else
    c = -1

cmp    dword[a], 10              ; compute a - 10, set flags
jnl    else                      ; if not less than 10, go to else

then:   mov    dword[c], 3         ; a < 10, set c to 3
        jmp    done                 ; and skip else part

else:   mov    dword[c], -1        ; a >= 10, set c to -1

done:    . . .
```
Another example with cmp

; int a, c    // signed ints
; if (a < 10)
;     c = 3
; else
;     c = -1

cmp    dword[a], 10  ; compute a - 10, set flags
j1     then            ; if a<10 go to then

else:  mov    dword[c], -1  ; otherwise, a>=10, set c to -1
        jmp    done  ; and skip then part

then:  mov    dword[c], 3   ; a < 10, set c to 3

done:   . . .
Exercise 1

Print array of 0s and 1s as series of ' ' or '#' symbols

```
dish db 0,1,1,0,0,0,0,0,0,0,0
N equ $-dish
```
Exercise 2

Read a string of '.' and '#' from user and transform it into an array of 0s and 1s

db    0          ; pad
dish   db  0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0
N      equ  $-dish
db    0          ; pad

call  _getString  ; buffer addr in ecx
             ; no. chars in edx
Exercise 3

Read dish of cells, and apply fate to each one: 1 neighbor alive, live in next generation, else die.

db 0 ; pad
dish db 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
N equ $ - dish
db 0 ; pad
newGen db 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

call _getString ; buffer addr in ecx
 ; no. chars in edx
Exercise 4

Compute and print fibonacci terms until the terms become larger than 1000.

```
%define fibn    eax
%define fibn_1  ebx
%define fibn_2  ecx

mov    fibn, 1
mov    fibn_1, 1

mov    eax, 1
call   _printInt ;print first 2 terms
call   _println

forLoop:
mov    fibn_2, fibn_1 ;fib(n-2) <- fib(n-1)
mov    fibn_1, fibn   ;fib(n-1) <- fib(n)
add    fibn, fibn_2   ;fib(n) <- fib(n-1)+fib(n-2)
```
%define fibn    eax
%define fibn_1  ebx
%define fibn_2  ecx

mov    fibn, 1
mov    fibn_1, 1

mov    eax, 1
call   _printInt ;print first 2 terms
call   _println
call   _printInt
call   _println

forLoop:
    mov    fibn_2, fibn_1 ;fib(n-2) <- fib(n-1)
    mov    fibn_1, fibn   ;fib(n-1) <- fib(n)
    add    fibn, fibn_2   ;fib(n) <- fib(n-1)+fib(n-2)

What about all the fibonacci terms less than 1,000,000,000? Less than 2,000,000,000? Less than 3,000,000,000?