Life Helpsession

"Life is far too important a thing ever to talk seriously about."
- Oscar Wilde

CS31
Brian Moore

The Algorithm

A high level view (from the handout):

```
initialize the board
for # of generations
  for each cell
    count neighbors
    determine next generation state
  print board
  swap arrays
```

and:

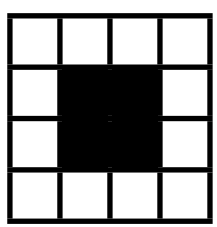
- A living cell with 2 or 3 live neighbors will stay alive, otherwise it will die.
- A dead cell with exactly three neighbors will become a live cell in the next generation, otherwise it will stay dead.

Swapping

- Why do we need to swap arrays?
  - all births and deaths happen at the same time
  - modifications to the life world cannot be done in place (on the same array)
- You are not allowed to simply copy the array. How do you swap, then?
  - you will have to be clever about manipulating pointers (addresses of variables in memory)
  - hint: consider how you can use pointers to access arrays indirectly (loads and stores)

Special Cases

- corners and borders
  - grid ends, fewer neighbors than usual
  - simply count the neighbors "off the grid" as dead
- test thoroughly
  - try various initial conditions
    - ex) an isolated 2x2 box of alive cells → should not change between generations



"Dost thou love life? Then do not squander time, for that the stuff life is made of."
- Benjamin Franklin

A Memory Refresher

- memory can be visualized as a sequence of fixed-size cells
- pointers are simply addresses which refer to a specific cell in memory
- will be using arrays (pointer arithmetic)
  - must keep in mind the size of cells

Memory in MIPS

- `lw`: loads one word (32 bits, 4 bytes) of data from the given address in RAM into the given register
  - the given address must be word-aligned (divisible by 4) - spim will raise exception otherwise
- `lb`: similar to `lw`, but only loads one byte and addresses do not have to be word-aligned
- `la`: will load the 32-bit address (in RAM) of the given variable into the given register. (for example, if foo is at memory location 0x1003, "`la $a0 foo`" will make register a0 contain 0x1003)

Memory in MIPS

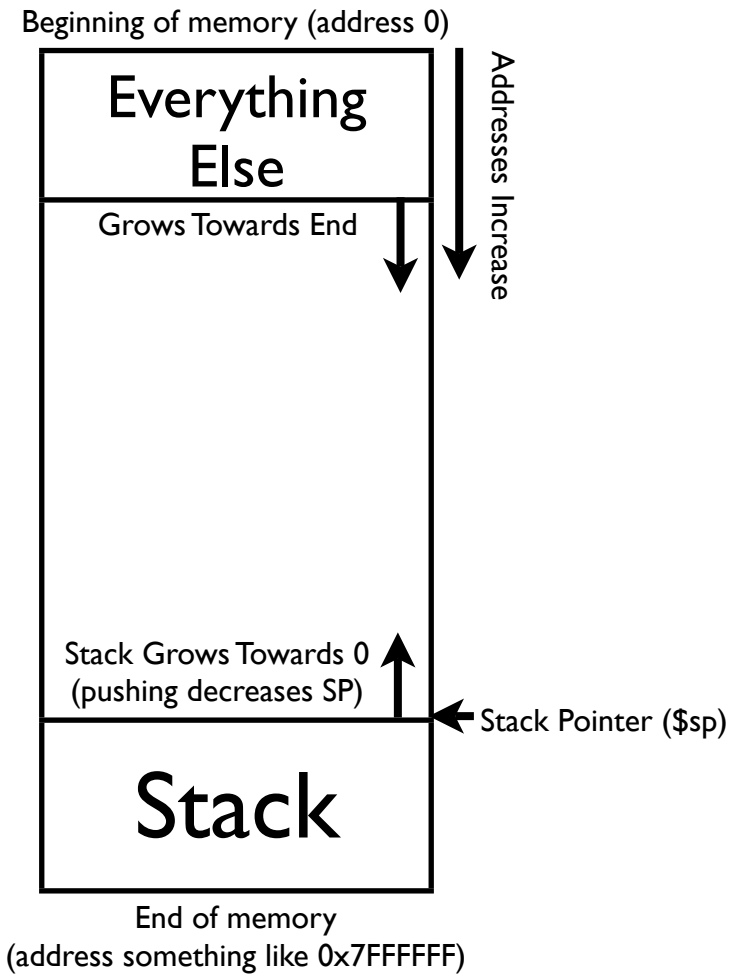
- `sw`: writes the 32-bit (1 word) contents of the given register to RAM at the given address (which must be word-aligned)

- `sb`: similar to `sw`, but writes the 8 least significant bits of the given register to RAM. Again, the address is not required to be word-aligned.

"Life is like an onion: You peel it off one layer at a time, and sometimes you weep."
- Carl Sandburg

Stack Management

- the stack is at the *end* of memory
  - everything else (including the "heap") is at the beginning of memory
  - stack grows towards zero - the beginning of memory
    - sometimes said to grow "down", which assumes a diagram rotated 180° from this one (with address 0 at the bottom and the end of memory at the top)
  - the stack pointer (SP) is the address just above the top of the stack
    - top element is just *below* SP (at SP+4)
  - pushing data onto the stack will decrease SP (moves up in this diagram), popping will increase SP



Pushing

- To push a value onto the stack, the following must happen:
  1. decrease the stack pointer (in $sp)
     - this reserves space for what you are pushing
  2. write the value to the stack
     - use MIPS offset notation

- for example:

```
sub $sp, $sp, 8  # reserve space on the top of the stack for two
                 # words (the size of two registers) by decreasing
                 # SP by 8 bytes
sw  $s0,4($sp)     # save s0 on the top of the stack
sw  $s1,8($sp)   # save s1 on the stack directly under the top
                 # (at address SP+8)
```

Popping

- To pop a value off of the stack, the following must happen:
  1. read values from the stack
     - use offset notation
  2. increase SP to reclaim the space which the value took up

- for example, to restore the values we pushed before:

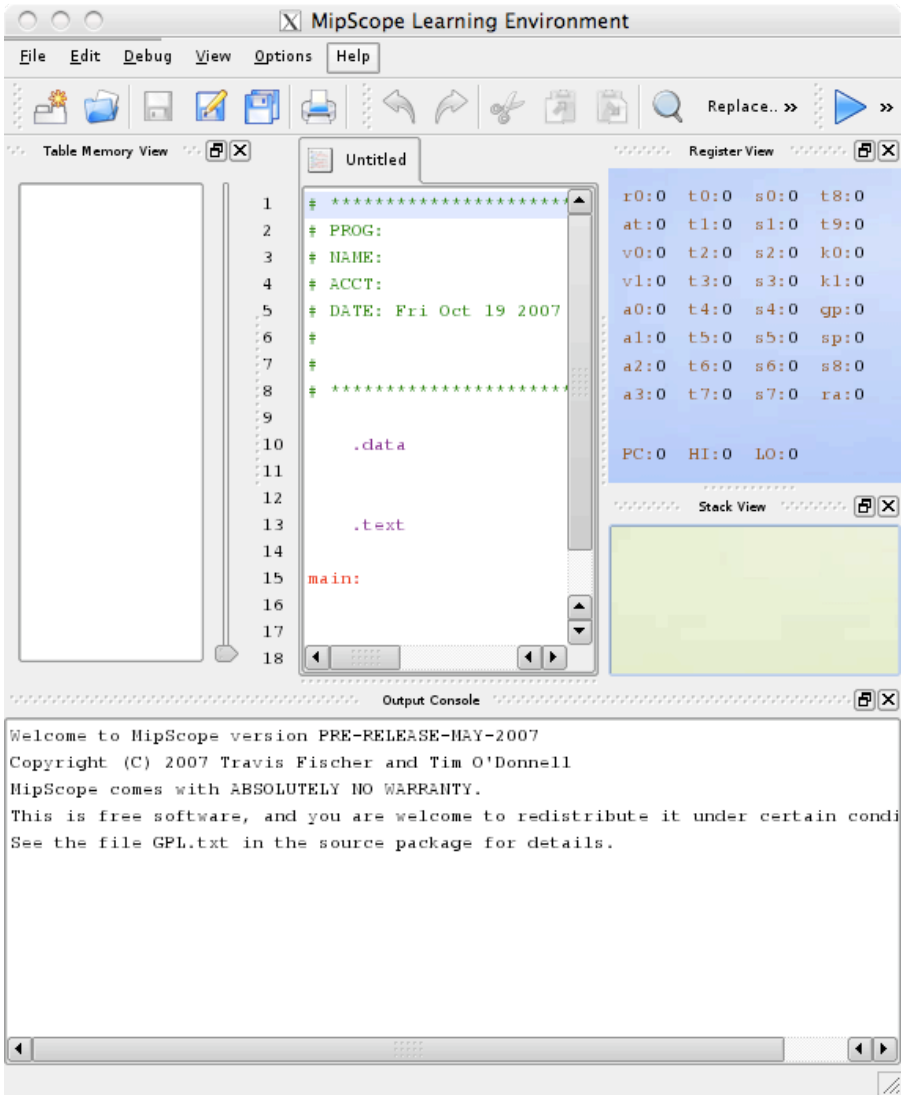
```
lw  $s0,4($sp)   # read s0 from the top of the stack
lw  $s1,8($sp)   # read s1 from the location directly under the
                 # top of the stack (at address SP+8)
add $sp, $sp, 8  # reclaim two words of space by adding 8 bytes to
                 # the stack pointer
```

- See the lecture on procedures for more examples

“The unexamined life is not worth living.”
-Socrates

Mipscope

- Makes writing and debugging your MIPS programs easier
  - written by two of last year's students
  - still a work in progress, please tell us about bugs or possible improvements
- Allows you to set breakpoints in your assembly where execution will automatically pause
  - You can even step your program backwards instruction by instruction



"Life is as tedious as a twice-told tale
Vexing the dull ear of a drowsy man."

- Shakespeare

Miscellaneous Tips

- Beyond not copying the entire life world array, don't worry too much about efficiency
- Write grid printing routine early on
- To print newlines, have the following in your .data section:

```
newline_str: .asciiz "\n"
```

  and use syscall 4 to print newline_str whenever you need a newline

More Miscellaneous Tips

- Just like for procedure calls, only s-registers (s0-s7) are guaranteed to be preserved across syscalls, all other registers may change.
  - ex) after the execution of

```
li $t0,13
li $s0,7
# print 42 using syscall 1
li $a0,42
li $v0,1
syscall
```

s0 will still contain 7, but t0 may no longer contain 13

Questions?

Some Motivation to Get You Started...

- "Attack life, it's going to kill you anyway."

  - Steven Coallier

- "Life is full of misery, loneliness, and suffering - and it's all over much too soon."

  - Woody Allen

- "The supreme irony of life is that hardly anyone gets out of it alive."

  - Robert Heinlein

- "Life is at best a dream and at worst a nightmare from which you cannot escape."

  - Mark Twain

Have fun!