Lecture 9: Assembly Programming Part 2

Last Week

- Assembly basics
  - QtSpim
- ALU operations
  - Arithmetic
  - Logical
  - Shift
- Branches (conditions and loops)
- Pseudoinstructions

```
addi $t6, $zero, 10
add $t6, $t6, $t1
add $t6, $t6, $t1
mult $t0, $t0
mflo $t4
add $t4, $t4, $t6
```

```
main:  add $t0, $0, $0
       addi $t1, $0, 100
LOOP : beq $t0, $t1, END
       addi $t0, $t0, 1
       j LOOP
END:
```

Homework

- Fibonacci sequence:
  - How would you convert this into assembly?

```
int n = 10;
int f1 = 1, f2 = 1;

while (n != 0) {
   f1 = f1 + f2;
   f2 = f1 - f2;
   n = n - 1;
}
# result is f1
```

Assembly code example

- Fibonacci sequence in assembly code:

```
# fib.asm
# register usage: $t3=n, $t4=f1, $t5=f2
#
FIB:  addi $t3, $zero, 10      # initialize n=10
      addi $t4, $zero, 1       # initialize f1=1
      addi $t5, $zero, 1       # initialize f2=-1
LOOP: beq $t3, $zero, END      # done loop if n==0
      add $t4, $t4, $t5        # f1 = f1 + f2
      sub $t5, $t4, $t5        # f2 = f1 - f2
      addi $t3, $t3, -1        # n = n - 1
      j LOOP                   # repeat until done
END:                           # result in f1 = $4
```

Making sense of assembly code

- Assembly language looks intimidating because the programs involve a lot of code.
  - No worse than your CSCA08 assignments would look to the untrained eye!
- The key to reading and designing assembly code is recognizing portions of code that represent higher-level operations that you're familiar with.

Interacting With Memory

Interacting with memory

- All of the previous instructions perform operations on registers and immediate values.
  - What about memory?
- All programs must fetch values from memory into registers, operate on them, and then store the values back into memory.
- Memory operations are I-type, with the form:

```
lw  $t0, 12($s0)
```

Load or store

Local data register

Offset from memory address

Register storing address of data value in memory

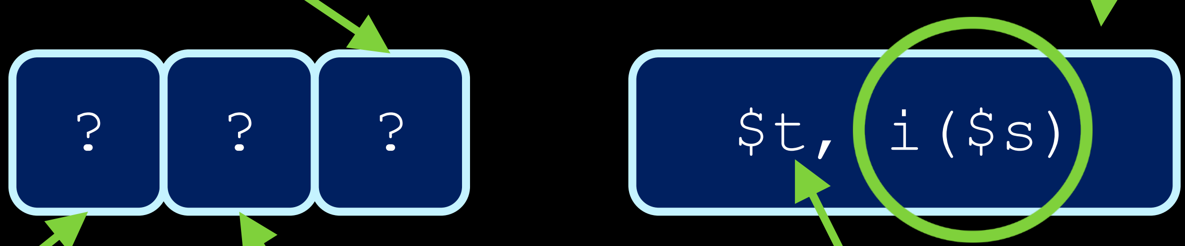
Loads vs. Stores

- The terms “load” and “store” are seen from the perspective of the processor, looking at memory.
- Loads are read operations.
  - We load (i.e., read) from memory.
  - We load a value **from** a memory address into a register.
- Stores are write operations.
  - We store (i.e., write) a data value from a register **to** a memory address.
  - Store instructions do not have a destination register, and therefore do not write to the register file.

Memory Instructions in MIPS assembly

When loading a byte or a half-word you can choose u for unsigned. Leave it blank as for all other cases.

Specifies the location to access as MEM[$s + SE(i)]



l for load or
s for store

b for byte,
h for half-word,
w for word

Destination register for loads, source register for stores.

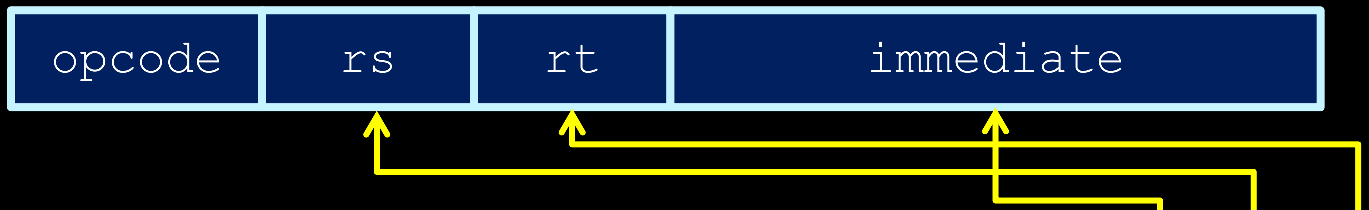
Load & store instructions

Instruction	Opcode/Function	Syntax	Operation
`lb`	100000	$t, i ($s)	$t = SE (MEM [$s + i]:1)
`lbu`	100100	$t, i ($s)	$t = ZE (MEM [$s + i]:1)
`lh`	100001	$t, i ($s)	$t = SE (MEM [$s + i]:2)
`lhu`	100101	$t, i ($s)	$t = ZE (MEM [$s + i]:2)
`lw`	100011	$t, i ($s)	$t = MEM [$s + i]:4
`sb`	101000	$t, i ($s)	MEM [$s + i]:1 = LB ($t)
`sh`	101001	$t, i ($s)	MEM [$s + i]:2 = LH ($t)
`sw`	101011	$t, i ($s)	MEM [$s + i]:4 = $t

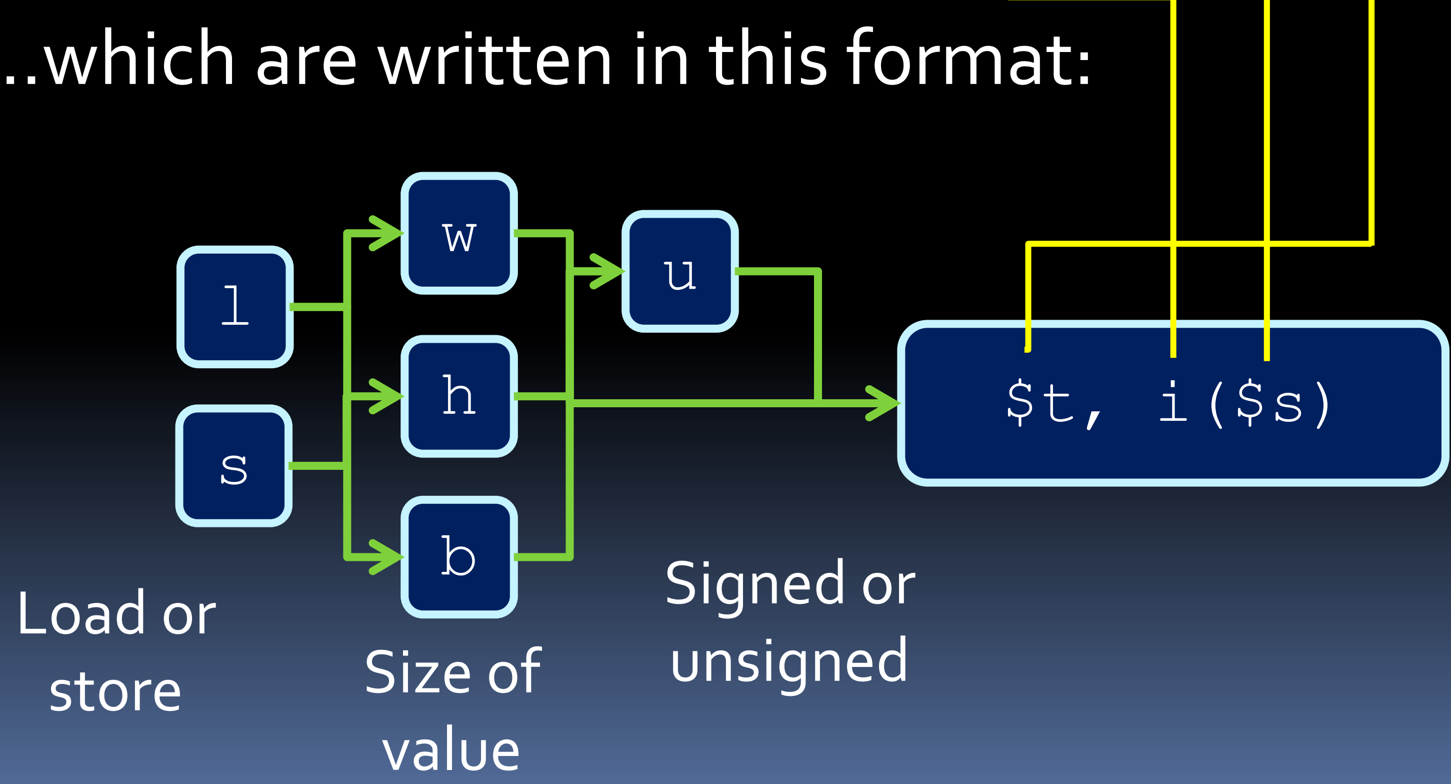
- "b", "h" and "w" correspond to "byte", "half word" and "word", indicating the length of the data.
- "SE" stands for "sign extend", "ZE" stands for "zero extend".

Memory Instructions in MIPS assembly

- Load & store instructions are I-type operations:



- ...which are written in this format:

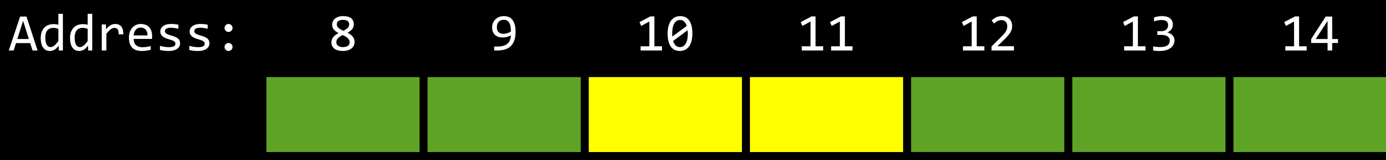


Alignment Requirements

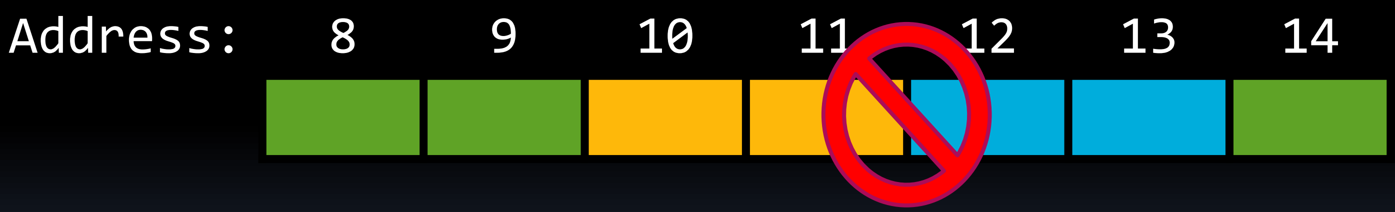
- Misaligned memory accesses result in errors.
  - Causes an exception (more on that, later)
- Word accesses (i.e., addresses specified in a `lw` or `sw` instruction) should be word-aligned (divisible by 4).
- Half-word accesses should only involve half-word aligned addresses (i.e., even addresses).
- No constraints for byte accesses.

Alignment Examples

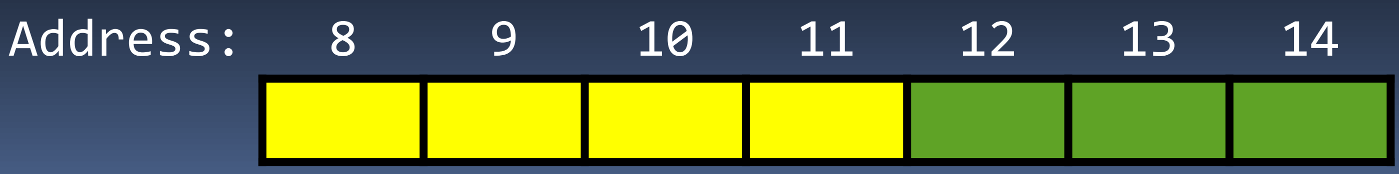
- Access to half-word at address 10 is aligned



- Access to word at address 10 is unaligned



- Access to word at address 8 is aligned



More Pseudo-instructions

Instruction	Opcode/Function	Syntax	Operation
`la`	N/A	$t, label	$t = address(MEM [label])
`li`	N/A	$t, i	$t = i

- Remember: these aren't really MIPS instructions
- But they make things way easier
- Really just simplifications of multiple instructions:
  - `lui` followed by `ori`. `$at` used for temporary values
- Also: `move`, `bge` , `ble`, `bgt`, `seq` …

Labeling Data Storage

- Labeled data storage, also known as variables
- At beginning of program, create labels for memory locations that are used to store values.
- Always in form: `label .type value(s)`

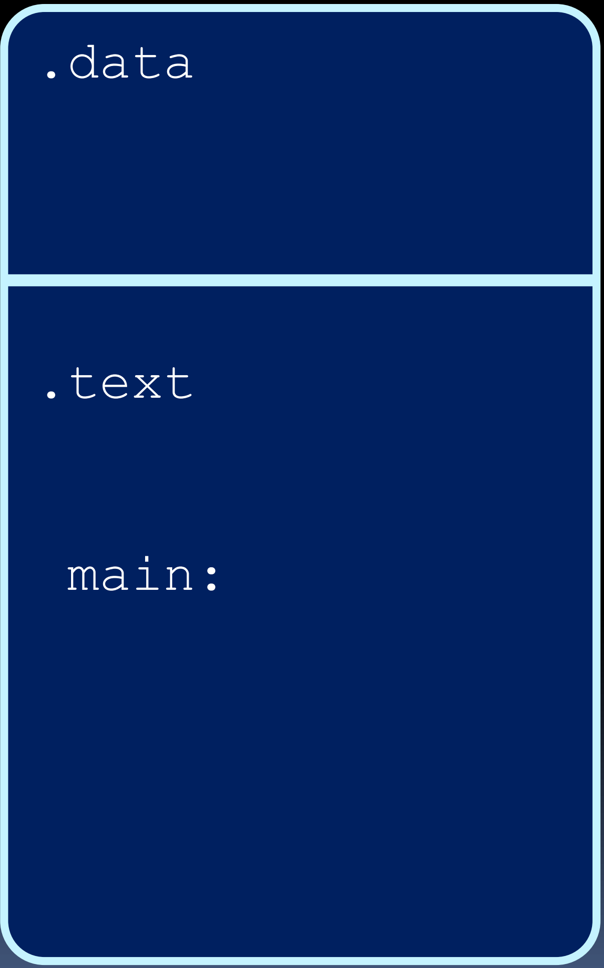
```
# create a single integer variable with initial value 3
var1:          .word     3
```

```
# create a 2-element character array with elements
# initialized to a and b
array1:        .byte    'a','b'
```

```
# allocate 40 consecutive bytes, with uninitialized
# storage. Could be used as a 40-element character
# array, or a 10-element integer array.
array2:        .space   40
```

Memory Sections & syntax

- Programs are divided into two main sections in memory:
- `.data` - indicates the start of the data values section (typically at beginning of program).
- `.text` - indicates the start of the program instruction section.
- Within the instruction section are program labels and branch addresses.
  - `main:` the initial line to run when executing the program.
  - Other labels are determined by the function names that you use, etc.



Arrays and Structs

Arrays!

- A sequence of data elements which is contiguous (i.e. no spaces) in memory.
- B is an array of 9 bytes starting at address 8:

Address:	8	9	10	11	12	13	14	15	16
	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]

- H is an array of 4 half-words starting at address 8:

Address:	8	9	10	11	12	13	14	15
	H[0]		H[1]		H[2]		H[3]	

Arrays

```
int A[100], B[100];
for (i=0; i<100; i++) {
   A[i] = B[i] + 1;
}
```

- Arrays in assembly language:
  - The address of the first element of the array is used to store and access the elements of the array.
  - To access element i in the array: start with the address of the first element and add an offset (distance) to the address of the first element.
    - offset = i * the size of a single element
    - address = address of first element + offset
  - Arrays are stored in memory. To process: load the array values into registers, operate on them, then store them back into memory.

Translating arrays

```c
int A[100], B[100];
for (i=0; i<100; i++) {
    A[i] = B[i] + 1;
}
```

```
.data
A:       .space 400          # array of 100 integers
B:       .word  42:100       # array of 100 integers, all
                             # initialized to value of 42
.text
main:    la $t8, A                  # $t8 holds address of array A
         la $t9, B                  # $t9 holds address of array B
         add $t0, $zero, $zero      # $t0 holds i = 0
         addi $t1, $zero, 100       # $t1 holds 100

LOOP:    bge $t0, $t1, END     # exit loop when i>=100
         sll $t2, $t0, 2       # $t2 = $t0 * 4 = i * 4 = offset
         add $t3, $t8, $t2     # $t3 = addr(A) + i*4 = addr(A[i])
         add $t4, $t9, $t2     # $t4 = addr(B) + i*4 = addr(B[i])
         lw $t5, 0($t4)        # $t5 = B[i]
         addi $t5, $t5, 1      # $t5 = $t5 + 1 = B[i] + 1
         sw $t5, 0($t3)        # A[i] = $t5
UPDATE:  addi $t0, $t0, 1      # i++
         j LOOP                # jump to loop condition check
END:     ...                   # continue remainder of program.
```

Optimization!

```
int A[100], B[100];
for (i=0; i<100; i++) {
   A[i] = B[i] + 1;
}
```

```
.data
A:        .space   400              # array of 100 integers
B:        .word    21:100           # array of 100 integers,
                             # all initialized to 21 decimal.


.text
main:     la $t8, A                 # $t8 holds address of A
          la $t9, B                 # $t9 holds address of B
          add $t0, $zero, $zero # $t0 holds 4*i; initially 0
          addi $t1, $zero, 400  # $t1 holds 100 * sizeof(int)

LOOP:     bge $t0, $t1, END   # branch if $t0 >= 400
          add $t3, $t8, $t0   # $t3 holds addr(A[i])
          add $t4, $t9, $t0   # $t4 holds addr (B[i])
          lw $t5, 0($t4)      # $t5 = B[i]
          addi $t5, $t5, 1    # $t5 = B[i] + 1
          sw $t5, 0($t3)      # A[i] = $t5
          addi $t0, $t0, 4    # update offset in $t0 by 4
          j LOOP
END:
```

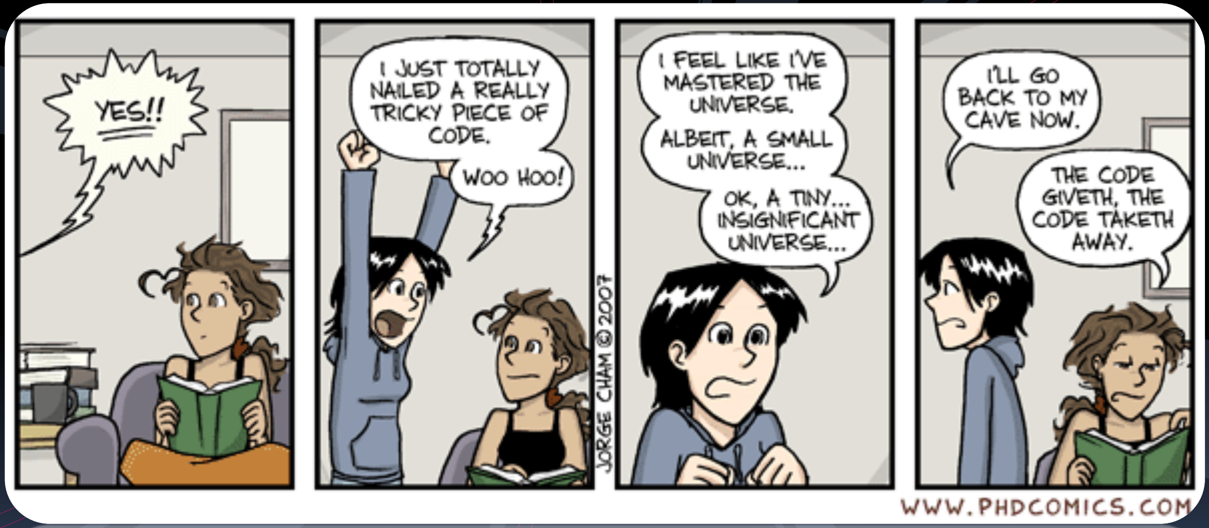
Yet Another Alternative

```
.data
A:        .space   400      # array of 100 integers
B:        .space   400      # array of 100 integers

.text
main:     add $t0, $zero, $zero     # load “0” into $t0
          addi $t1, $zero, 400      # load “400" into $t1
          addi $t9, $zero, B        # store address of B
          addi $t8, $zero, A        # store address of A

loop:     add $t4, $t8, $t0  # $t4 = addr(A) + i
          add $t3, $t9, $t0  # $t3 = addr(B) + i
          lw $s4, 0($t3)     # $s4 = B[i]
          addi $t6, $s4, 1   # $t6 = B[i] + 1
          sw $t6, 0($t4)     # A[i] = $t6
          addi $t0, $t0, 4   # $t0 = $t0++
          bne $t0, $t1, loop # branch back if $t0<400
end:
```

Break



YES!!
I JUST TOTALLY NAILED A REALLY TRICKY PIECE OF CODE.
WOO HOO!
JORGE CHAM © 2007
I FEEL LIKE I'VE MASTERED THE UNIVERSE.
ALBEIT, A SMALL UNIVERSE...
OK, A TINY... INSIGNIFICANT UNIVERSE...
I'LL GO BACK TO MY CAVE NOW.
THE CODE GIVETH, THE CODE TAKETH AWAY.
WWW.PHDCOMICS.COM

Structs

- Structs are simply a collection of fields one after another in memory
  - With optional padding so memory access are aligned
- Assembly does not understand structs
  - But load/store instructions allow fixed offset!

```
struct {
      int a;
      int b;
      int c;
} s;

s.a = 5;
s.b = 13;
s.c = -7;
```

Example: A struct program

- How can we figure out the main purpose of this code?
- The `sw` lines indicate that values in `$t1` are being stored at `$t0`, `$t0+4` and `$t0+8`.
  - Each previous line sets the value of `$t1` to store.
- Therefore, this code stores the values `5`, `13` and `-7` into the struct at location `a`.

```
            .data
s:          .space    12

            .text
main:       addi      $t0, $zero, s
            addi      $t1, $zero, 5
            sw        $t1, 0($t0)
            addi      $t1, $zero, 13
            sw        $t1, 4($t0)
            addi      $t1, $zero, -7
            sw        $t1, 8($t0)
```

Functions vs Code

- Up to this point, we've been looking at how to create pieces of code in isolation.
- A function creates an interface to this code by defining the input and output parameters.
- Once a function finishes, control returns to the caller, optionally with returned value.
- How can we do this in assembly?

Functions

- We can jump to a block of code and jump back
  - How do we know where to jump back to?
- Can complete functions that have no parameters or return value
  - Not very useful
  - How do we pass parameters and returned value?

Parameters: Option #1

- Reserve some registers for parameters & return values
- Look back at previous slides:
  - Registers 2-3 ($v0, $v1): return values
  - Registers 4-7 ($a0-$a3): function arguments
- Problems?
  - What if we need more parameters?
  - What if that function calls another function?
  - Recursion?

Parameters: Option #2

- Use a stack
- $sp register points to the top of the stack.
- Caller pushes parameters on top of stack (it grows)
- Function code pops the parameters from the stack using $sp.

Pushing on Stack

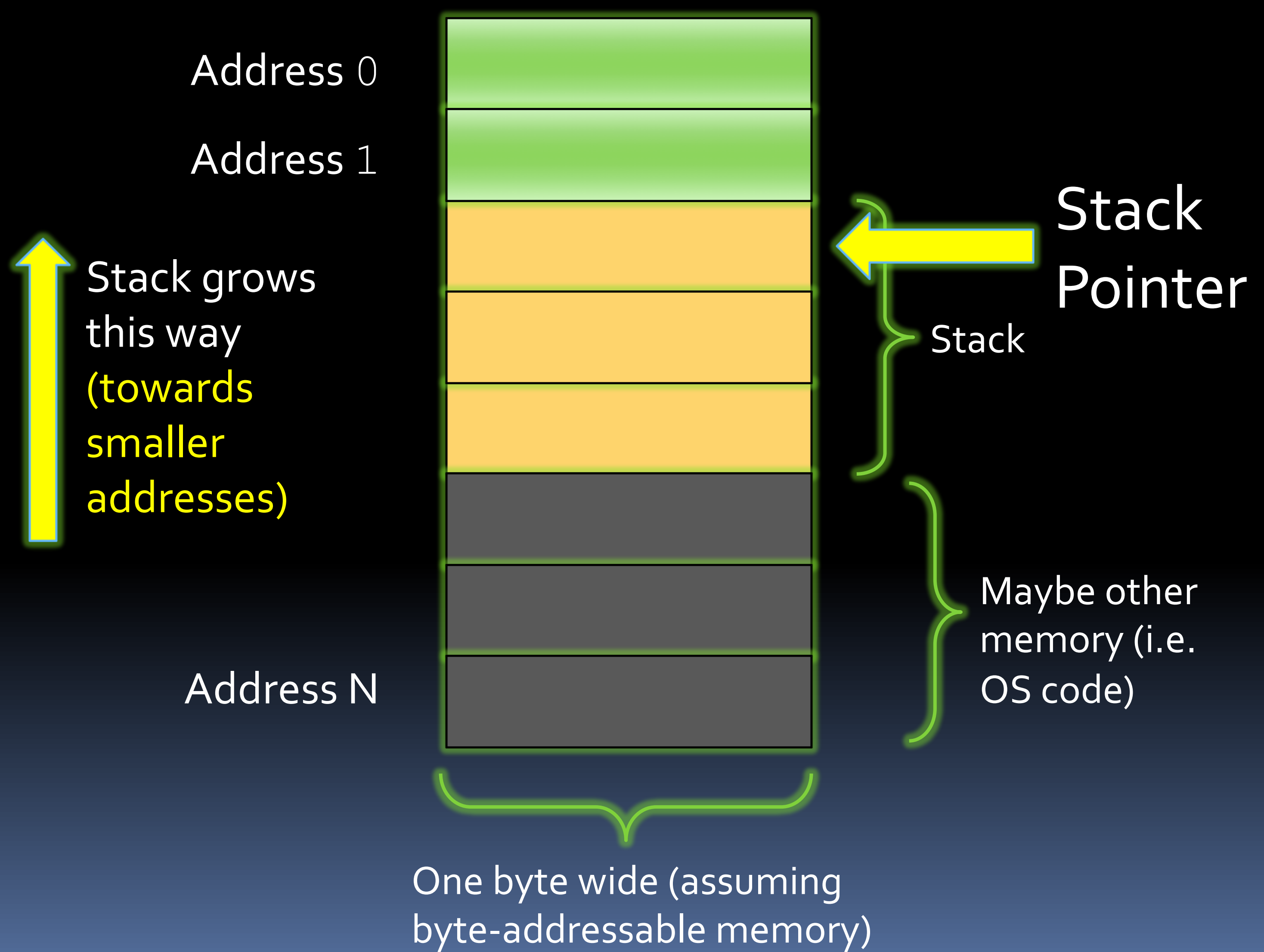
- Special register `$sp` stores the stack pointer
- PUSH value `$t0` onto the stack

```
addi  $sp, $sp, -4 # move stack pointer one word
sw    $t0, 0($sp)   # push a word onto the stack
```

- POP value from the stack onto `$t0`

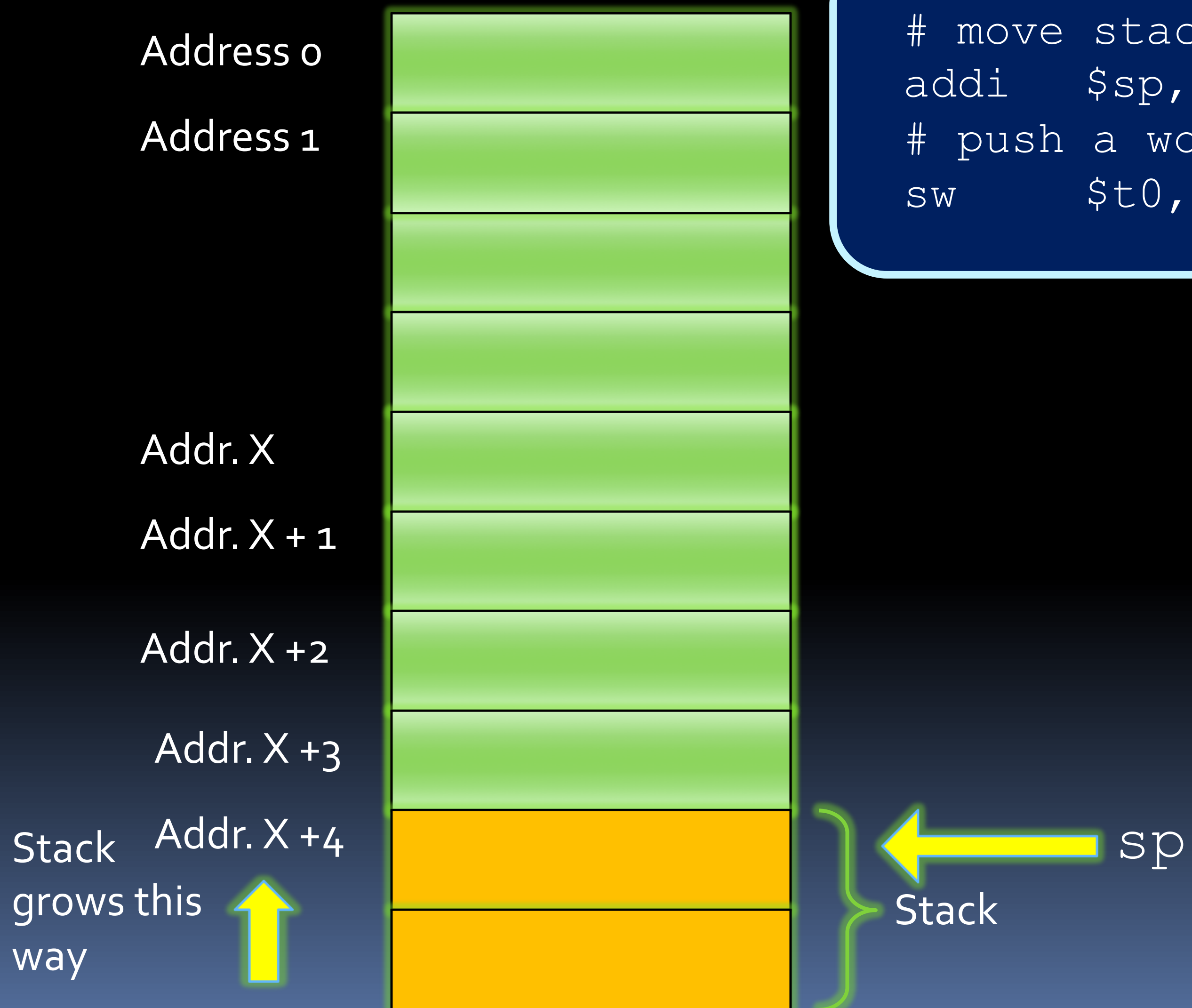
```
lw    $t0, 0($sp) # pop that word off the stack
addi  $sp, $sp, 4 # move stack pointer one word
```

The Stack, illustrated



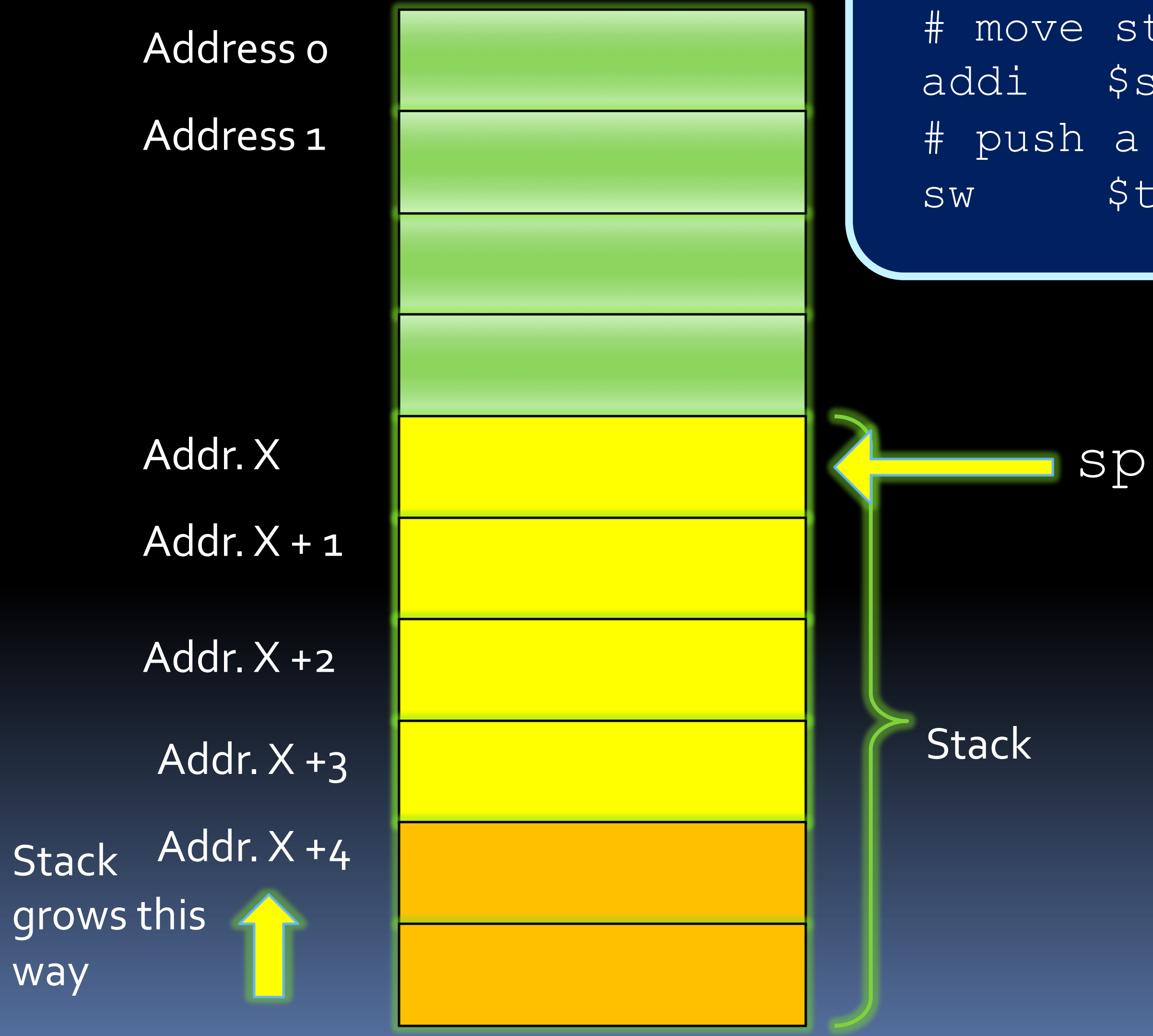
Pushing Values to the stack - Before

```
# move stack pointer a word
addi   $sp, $sp, -4
# push a word onto the stack
sw     $t0, 0($sp)
```



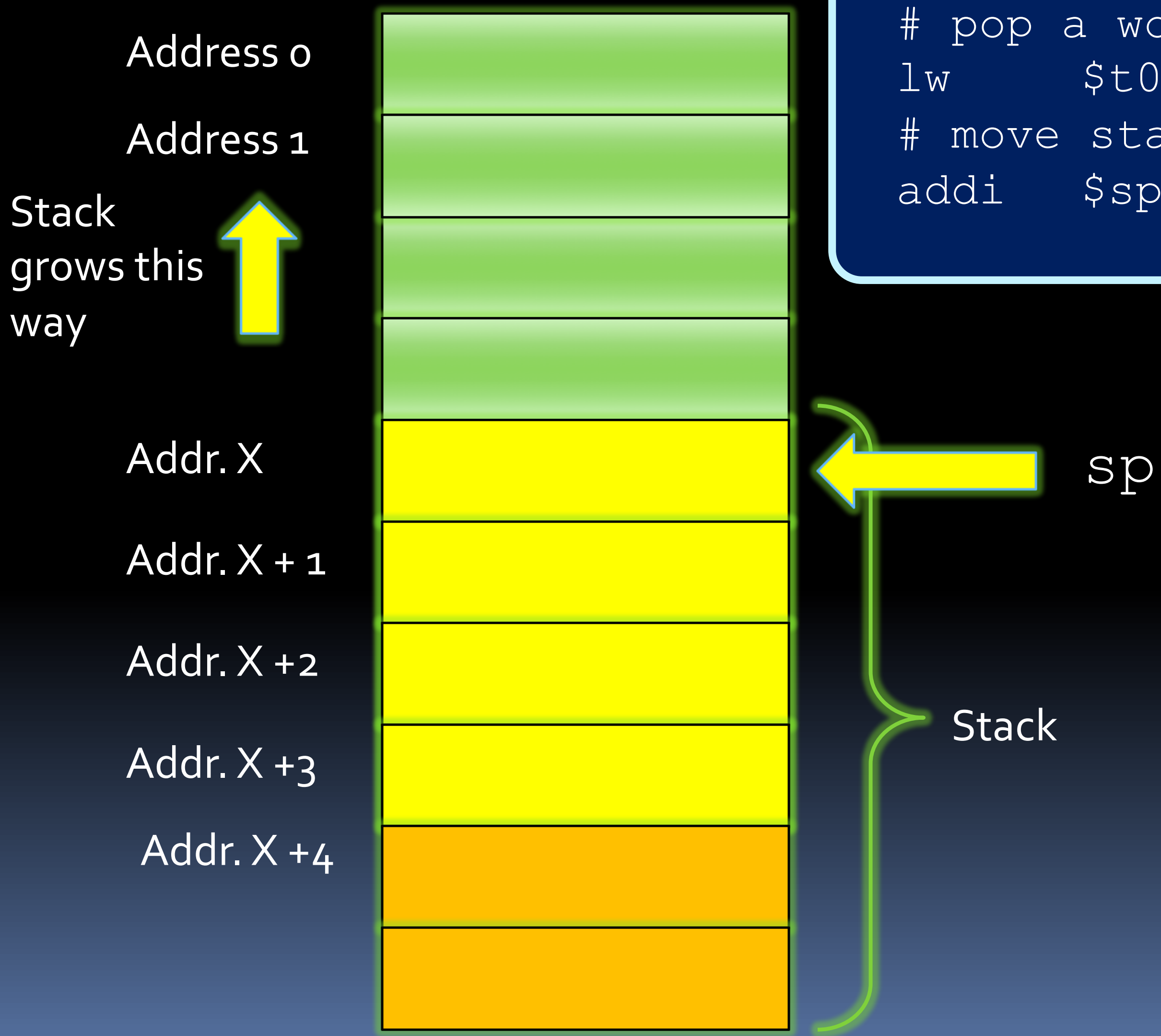
Pushing Values to the stack - After

```
# move stack pointer a word
addi   $sp, $sp, -4
# push a word onto the stack
sw     $t0, 0($sp)
```

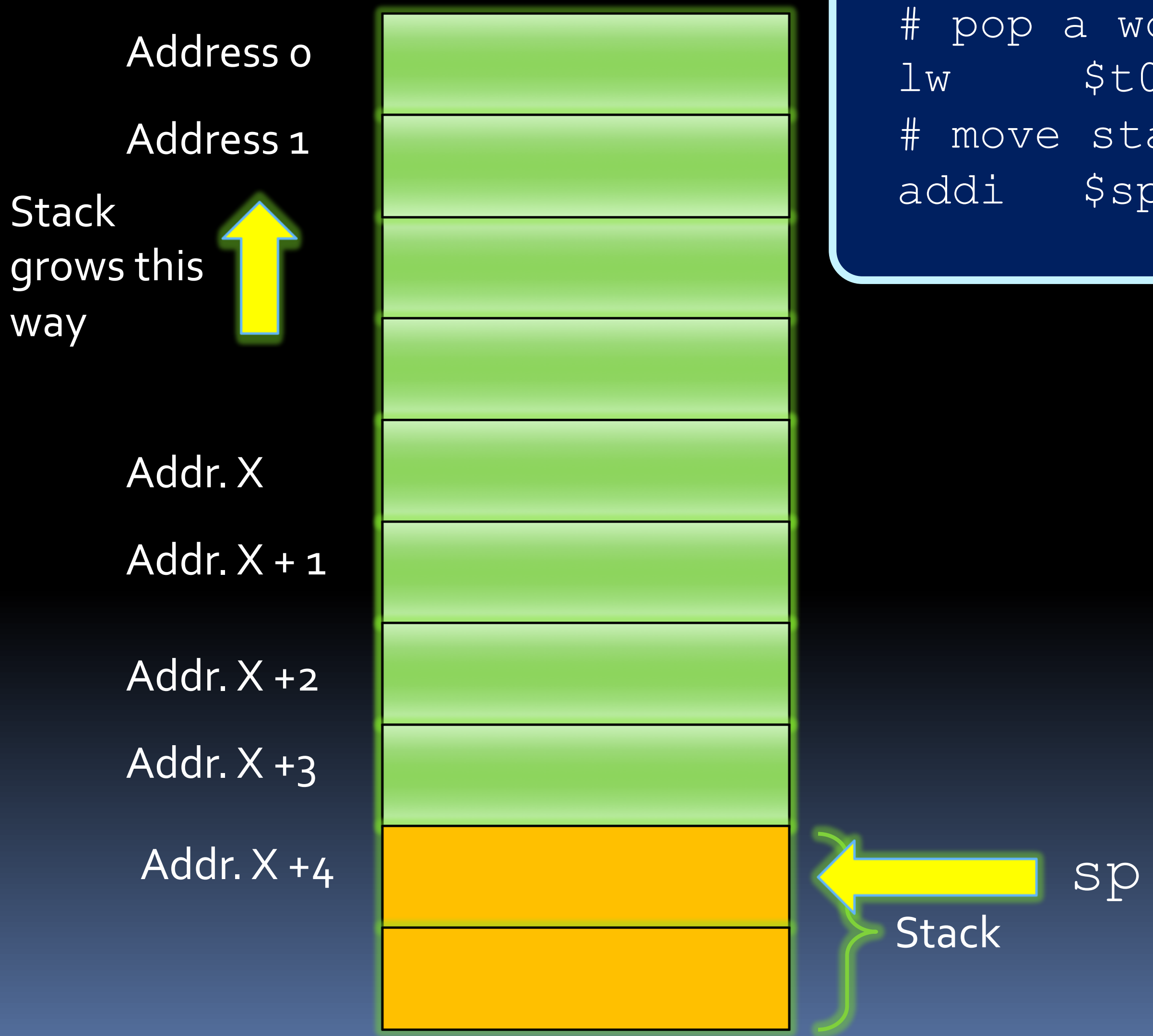


Popping Values off the stack - Before

```
# pop a word off the stack
lw     $t0,  0($sp)
# move stack pointer a word
addi   $sp, $sp, 4
```



Popping Values off the stack - After




```
# pop a word off the stack
lw      $t0,   0($sp)
# move stack pointer a word
addi   $sp, $sp, 4
```

String function program

```
void strcpy (char x[], char y[]) {
   int i;
   i=0;
   while ( (x[i] = y[i]) != 0 )
      i += 1;
   return 1;
}
```

Equivalent to `'\0'`

- Let's convert this to assembly code!
- Take in parameters from the stack
  - In this case, the parameters `x` and `y` are passed into the function, in that order.
- The pointer to the stack is stored in register `$29` (aka `$sp`), which is the address of *the top element of the stack*.

Converting strcpy()

Initialization:

- Parameters
  - Addresses of `x[0]` and `y[0]`
- We'll also need registers for:
  - The current offset value (`i` in this case)
  - Temporary values for the address of `x[i]` and `y[i]`
  - The current value being copied from `y[i]` to `x[i]`.

```
void strcpy (char x[], char y[]) {
    int i;
    i=0;
    while ((x[i] = y[i]) != 0)
        i += 1;
    return 1;
}
```

Converting strcpy()

- Initialization (cont'd):
  - Consider that the locations of `x[0]` and `y[0]` are passed in on the stack, we need to fetch those first.
  - Basic code for popping values off the stack:

```
lw      $t0, 0($sp)   # pop that word off the stack
addi    $sp, $sp, 4   # move stack pointer by a word
```

  - Basic code for pushing values onto the stack:

```
addi    $sp, $sp, -4  # move stack pointer one word
sw      $t0, 0($sp)   # push a word onto the stack
```

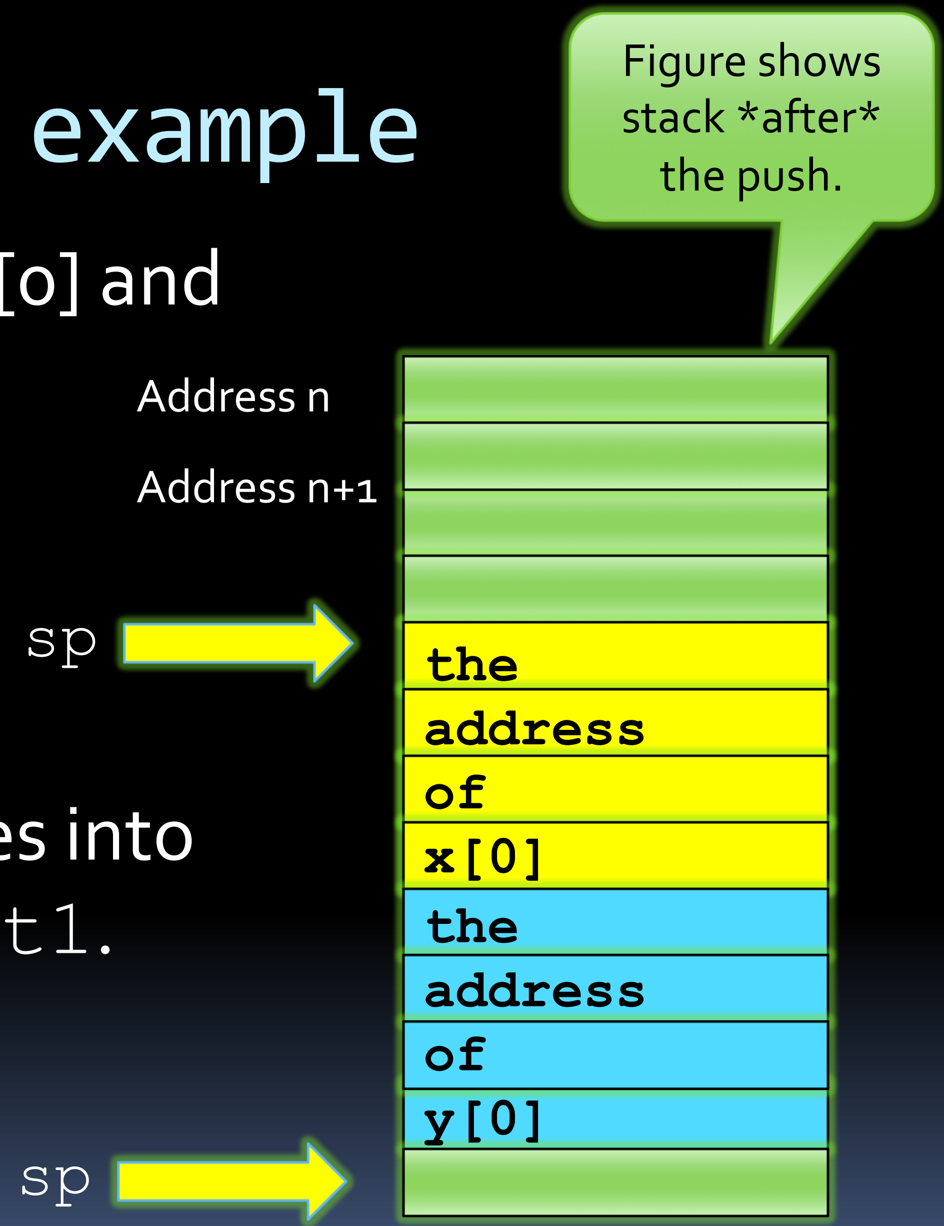
Stack storage example

- Push addresses of x[o] and y[o] onto the stack.

```
addi $sp, $sp, -8
sw $t0, 0($sp)
sw $t1, 4($sp)
```

- Pop stored addresses into registers `$t0` and `$t1`.

```
lw $t0, 0($sp)
lw $t1, 4($sp)
addi $sp, $sp, 8
```



Converting strcpy()

- Main algorithm: What steps do we need to perform?
  - Get the location of `x[i]` and `y[i]`.
  - Fetch a character from `y[i]` and store it in `x[i]`.
  - Jump to the end if the character is the NUL character.
  - Otherwise, increment `i` and jump to the beginning.
- At the end: push the value `1` onto the stack and return to the calling program.

```
void strcpy (char x[], char y[]) {
    int i;
    i=0;
    while ((x[i] = y[i]) != 0)
        i += 1;
    return 1;
}
```

Translated strcpy program

```
strcpy:              lw     $a0, 0($sp)           # pop x address
                     addi   $sp, $sp, 4           #  off the stack
   initialization    lw     $a1, 0($sp)           # pop y address
                     addi   $sp, $sp, 4           #  off the stack
                     add    $t0, $zero, $zero     # $t0 = offset i
L1:                  add    $t1, $t0, $a0         # $t1 = x + i
                     lb     $t2, 0($t1)           # $t2 = x[i]
                     add    $t3, $t0, $a1         # $t3 = y + i
main algorithm       sb     $t2, 0($t3)           # y[i] = $t2
                     beq    $t2, $zero, L2        # y[i] = '\0'?
                     addi   $t0, $t0, 1           # i++
                     j      L1                    # loop
L2:                  addi   $sp, $sp, -4          # push 1 onto
                end  addi   $t0, $zero, 1         #  the top of
                     sw     $t0, 0($sp)           #  the stack
                     jr     $ra                   # return
```

Calling Functions

- So we can pass parameters and return values by using the stack
- How do we know where to jump back to after function is done?
  - Could just put PC onto stack
  - Better option: Special register $ra = return address
  - Special operation: jal = jump and link
  - Jumps, and puts value of PC into $ra

How do we call a function?

- `jal FUNCTION_LABEL`
  - We do this **after** we've set the appropriate values to $a0-$a3 registers and/or pushed arguments to the stack.

```
…
sum = 3;
function_X(sum);
sum = 5;
```

- `jal` is a J-Type instruction.
  - It updates register `$31` (`$ra`, return address register) and also the Program Counter.
  - After it's executed, `$ra` contains the address of the instruction ***after*** the line that called `jal`.

How do we return from a function?

- `jr $ra`
  - The PC is set to the address in `$ra`.
- But how do we know what's in `$ra`?
  - `$ra` was set by the most recent `jal` instruction (function call)!

```
…
sum = 3;
function_X(sum);
sum = 5;
```

```
void function_X (int sum) {

   //do something

   return;
}
```

Function Calls – Cont'd

```
…
sum = 3;
function_X(sum);
sum = 5;
```

(1) `jal FUNCTION_X`
`$ra` set to PC of the next instruction

(4) Execution continues here

(2) Execution continues from here

```
void function_X (int sum) {

   //do something

   return;
}
```

(3) `jr $ra`

Putting it Together

- Caller calls Callee
  1. Caller pushes arguments onto the stack
  2. Caller stores current PC into $ra, jumps to Callee
  3. Callee pops arguments from the stack
  4. Callee performs function
  5. Callee pushes return value onto stack
  6. Callee jumps to address stored in $ra
  7. Caller pops return value from stack
  8. Caller continues on its marry way

Calling Conventions

- We've seen at least two options on how to implement function calls:
  - Use $a0 - $a3 , $v0 and $v1, and so on.
  - Push on stack
- There are many other variants.
  - For example, should caller or callee pop variables?
  - Or using registers instead of stack.
- These are called calling conventions.

Common Calling Conventions

- It is also possible to use registers to pass values to and from programs:
  - Registers `2–3` ($v0, $v1): return values
  - Registers `4–7` ($a0-$a3): function arguments
- If your function has up to 4 arguments, you would use the `$a0` to `$a3` registers in that order. Any additional arguments would be pushed on the stack.
  - First argument in `$a0`, second in `$a1`, and so on.
- For us: push all arguments and return values to the stack and pop them when needed.
  - We'll tell you if we want otherwise.

You Think it's Over?

Next week – more on functions:

- Local variables
- Saving registers
- Recursion
- Exceptions
- System calls
- Human sacrifice
- Dogs and cats living together
- Mass hysteria!