

CSC258 Week 10

Recap

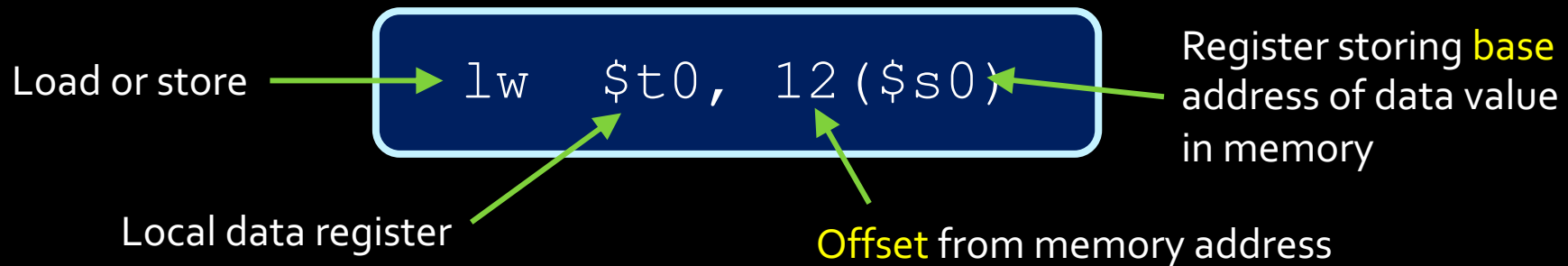
- We started learning about assembly, instruction by instruction
- Arithmetic operations
- Logical operations
- Branching
- Jump
- Comparison
- How to implement if-else and loops

Only a few more
instructions left!

- Memory access
- System calls

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:



Quick reminder

Word: 4-byte

Half-word: 2-byte

Byte: 1-byte

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.
- LB: lowest byte; LH: lowest half word

Examples

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

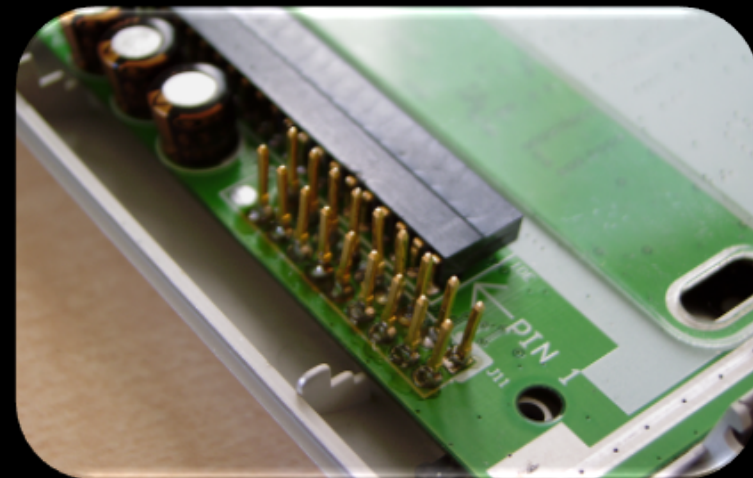
Load a **half-word** (2 bytes) starting from **MEM(\$s0 + 12)**, **sign-extend** it to 4 bytes, and store in \$t0

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

Take the **lowest byte** of the word stored in \$t0, store it to memory starting from address \$s0 + 12

A bit more about memory

- The **base + offset** style is useful for arrays or stack parameters, when multiple values are needed from a given memory location.
- Memory is also used to communicate with outside devices, such as keyboards and monitors.
 - Known as **memory-mapped IO**.
 - Invoked with a **trap** or function.





It's a
Trap!

Instruction	Function	Syntax
trap	011010	i

- **Trap instructions** send system calls to the operating system
 - e.g. interacting with the user, and exiting the program, raise exceptions.
- Similar to the `syscall` command.
 - use registers `$a0` and `$v0`

`$4` is `$a0`, `$2` is `$v0`

Service	Trap Code	Input/Output
print_int	1	\$4 is int to print
print_float	2	\$f12 is float to print
print_double	3	\$f12 (with \$f13) is double to print
print_string	4	\$4 is address of ASCIIZ string to print
read_int	5	\$2 is int read
read_float	6	\$f12 is float read
read_double	7	\$f12 (with \$f13) is double read
read_string	8	\$4 is address of buffer, \$5 is buffer size in bytes
sbrk	9	\$4 is number of bytes required, \$2 is address of allocated memory
exit	10	
print_byte	101	\$4 contains byte to print
read_byte	102	\$2 contains byte read
set_print_inst_on	103	
set_print_inst_off	104	
get_print_inst	105	\$2 contains current status of printing instructions

syscall example

```
li $v0, 4          # $v0 stores syscall number, 4 is print_string  
la $a0, promptA   # $a0 stores the address of the string to print  
syscall           # check $v0 and $a0 and act accordingly
```

Arrays and Structs

Data storage

- At the beginning of the program, create labels for memory locations that are used to store values.
- Always in form: **label** **type** **value**

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

# create a 4-element integer array
array0:        .word      3, 7, 5, 42

# 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
```

Integer type (int): 4 byte

Character type (char): 1 byte



Arrays and Structs

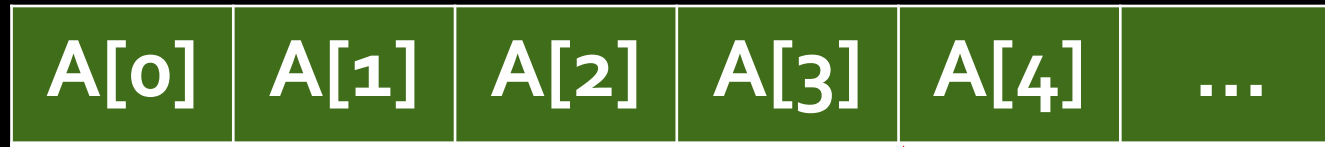


Arrays!

```
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 an element of the array, get the address of that element by adding an **offset** distance to the address of the first element.
 - **offset = array index * the size of a single element**
 - Arrays are stored in memory. For examples, fetch the array values and store them in registers. Operate on them, then store them back into memory.


```
int A[100];
```



Offset = 4×4 bytes = 16 bytes

Address of $A[4]$ = Address of $A[0]$ + 16 (bytes)

Translate this to assembly

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

Making sense of assembly code

- The key to reading and designing assembly code is recognizing **portions of code** that represent **higher-level operations** that you're familiar with.

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

```
.data
A:      .space 400
B:      .space 400

.text
main:   add $t0, $zero, $zero
        addi $t1, $zero, 400
        la $t8, A
        la $t9, B

loop:   add $t4, $t8, $t0
        add $t3, $t9, $t0
        lw $s4, 0($t3)
        addi $t6, $s4, 1
        sw $t6, 0($t4)
        addi $t0, $t0, 4
        bne $t0, $t1, loop

end:
```

Initialization:

- Allocate **space**
- Initial value **i=0 (offset)**, put into a register
- Put value **size (400)** in register
- Put **addresses** of A, B into register

The loop:

- Put **addrs** of A[i] and B[i] into registers ($\text{addr}(A)+\text{offset}$).
- **Load** B[i] from mem, then **+1**, keep result in a register
- Store result into mem A[i]
- Update i++
- Check loop condition and jump

Code with comments

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

```
.data  
A:      .space 400      # array of 400 bytes (100 ints)  
B:      .space 400      # array of 400 bytes (100 ints)  
  
.text  
main:   add $t0, $zero, $zero      # load "0" into $t0  
        addi $t1, $zero, 400      # load "400" into $t1  
        la $t9, B                  # store address of B  
        la $t8, 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:
```

Struct

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, 42 and 12 into the struct at location `a1`.

```
.data
a1:    .space    12

.text
main:  la        $t0, a1
      addi     $t1, $zero, 5
      sw      $t1, 0($t0)
      addi     $t1, $zero, 42
      sw      $t1, 4($t0)
      addi     $t1, $zero, 12
      sw      $t1, 8($t0)
```

Example: A struct program

```
struct foo {  
    int a;  
    int b;  
    int c;  
};  
  
struct foo x;  
x.a = 5;  
x.b = 42;  
x.c = 12;
```

```
        .data  
a1:     .space   12  
  
        .text  
main:  la      $t0, a1  
       addi   $t1, $zero, 5  
       sw    $t1, 0($t0)  
       addi   $t1, $zero, 42  
       sw    $t1, 4($t0)  
       addi   $t1, $zero, 12  
       sw    $t1, 8($t0)
```


Struct program with comments

```
.data
a1:      .space    12      # declare 12 bytes
                          # of storage to hold
                          # struct of 3 ints

.text
main:    la         $t0, a1 # load base address
                          # of struct into
                          # register $t0

        addi       $t1, $zero, 5 # $t1 = 5
        sw         $t1, 0($t0) # first struct
                          # element set to 5;
                          # indirect addressing

        addi       $t1, $zero, 42 # $t1 = 42
        sw         $t1, 4($t0) # second struct
                          # element set to 42

        addi       $t1, $zero, 12 # $t1 = 12
        sw         $t1, 8($t0) # third struct
                          # element set to 12
```

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

struct foo x;
x.a = 5;
x.b = 42;
x.c = 12;
```

Caveat: alignment

- Suppose `b` is a `char` (one byte), then it looks like that the address of `c` would be `$t0 + 5`.
- But this may not work because MIPS by default requires store addresses to be **word-aligned**, i.e., must be a multiple of 4.

```
struct foo {
    int a;
    char b;
    int c;
};
struct foo x;
x.a = 5;
x.b = 'B';
x.c = 12;
```

```
a1:      .data
        .space 12

main:    .text
        la     $t0, a1
        addi   $t1, $zero, 5
        sw     $t1, 0($t0)
        addi   $t1, $zero, 'B'
        sb     $t1, 4($t0)
        addi   $t1, $zero, 12
        ERROR sw     $t1, 5($t0)
```

Function calls

Another example:

A function!

Function arguments!

```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
}
```

Return!

```
int x, r;  
x = 42;  
r = sign(x);  
r = r + 1;  
...
```

Function arguments

```
int sign (int i) {
    if (i > 0)
        return 1;
    else if (i < 0)
        return -1;
    else
        return 0;
}

int x, r;
x = 42;
r = sign(x);
r = res + 1;
...
```

Where are the function arguments stored?

They are stored at a certain location in the **memory**, which is call the **stack**.

Other conventions are also possible, i.e., store first 4 arguments in \$a0~\$a3, the rest in the stack

Note

- Because assembly programmers have so much control over how things are done at the low level, there are always **multiple** ways of implementing a feature.
- We need to define a **convention** of how function arguments and return values are passed between functions, etc, so all programmers working on the same project are on the same page.
- There can be many different version of the conventions.

Memory model: a quick look



High address



Stack grows this way (going low)

If they collide



Heap grows this way (going high)

Low address

Note: stack grows **backwards**, i.e., when stack pointer (top) decreases, stack becomes bigger; when stack pointer increase, stack becomes smaller.

Function arguments

```
int sign (int i) {
    if (i > 0)
        return 1;
    else if (i < 0)
        return -1;
    else
        return 0;
}

int x, r;
x = 42;
r = sign(x);
res = r + 1;
...
```

Why keep the arguments in **memory** instead of **registers**?

Because there aren't enough registers for this

- One function may have many arguments
- If function calls subroutines, all subroutines' arguments need to be remembered. (can't forget until function returns)

Note

You can use the **registers** to store function arguments if you know you have enough registers to do so (e.g., one single-argument function with no subroutines).

An **assembly** programmer makes this type of design decisions and can do whatever they want.

For high-level language programmers, the **compiler** makes this type of decisions for them.

How to access stack?

The address of the “top” of the stack is stored in this register -- **\$sp**

PUSH value in \$t0 into stack

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

POP a value from stack and store in \$t0

```
lw      $t0, 0($sp)  # pop a word from the stack
addi    $sp, $sp, 4  # update stack pointer, stack size smaller
```

The Stack

Low address

Address 0

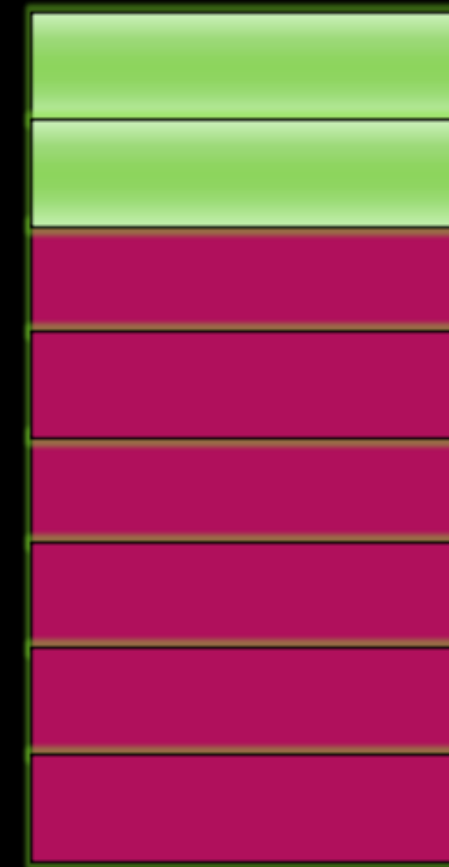
Address 1

Stack
grows this
way



Address N

High address

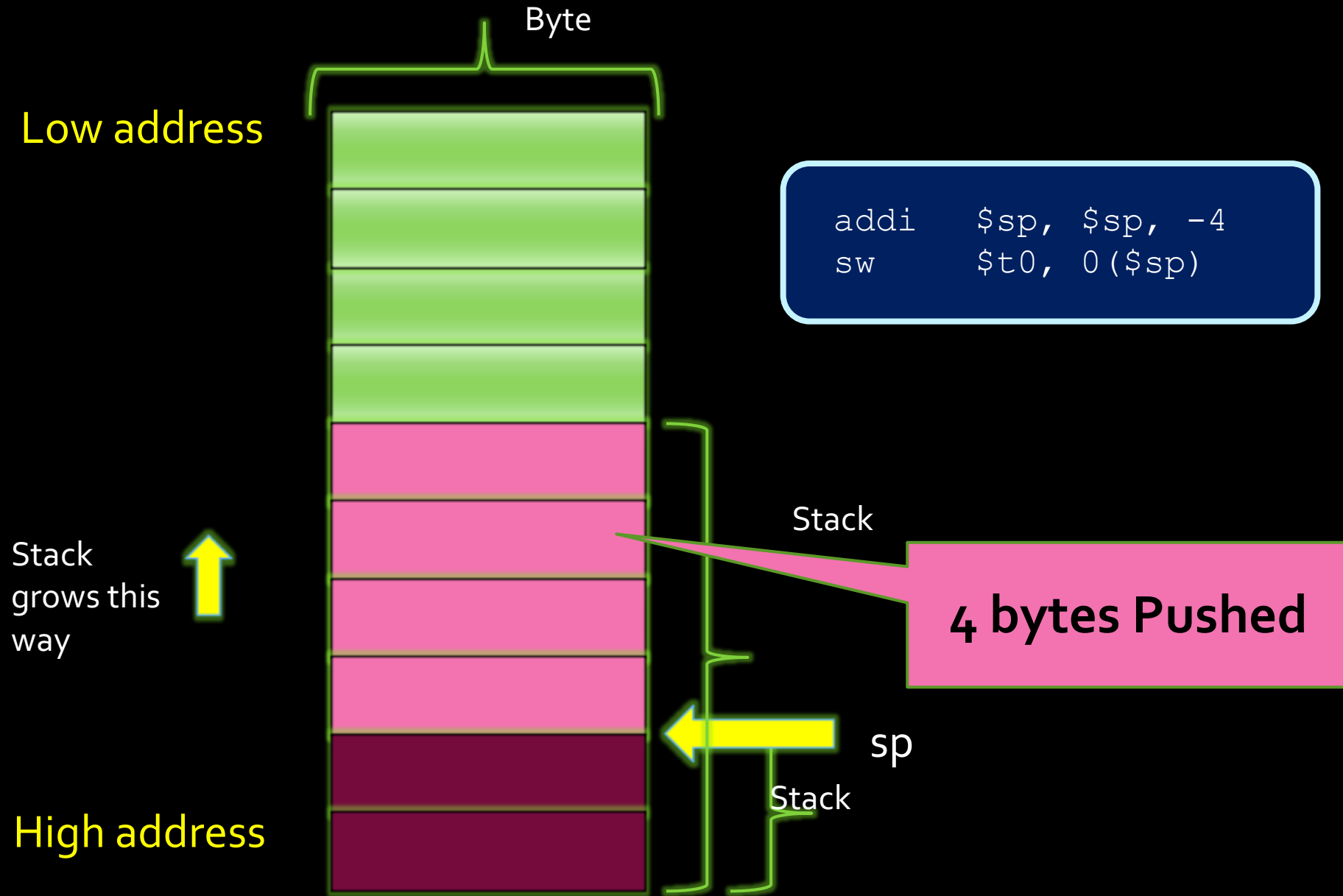


Stack
Pointer

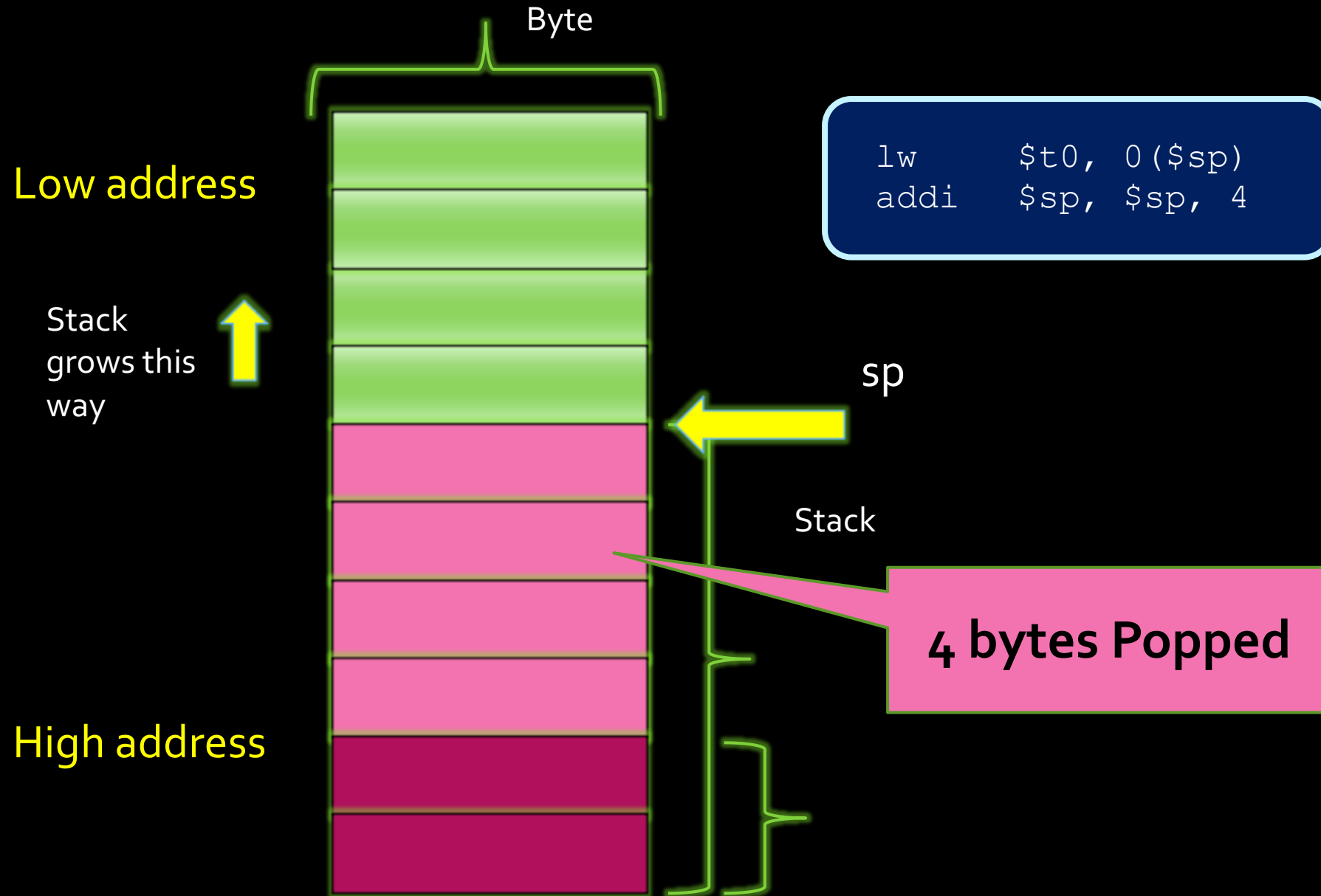
Stack

Byte

Pushing Values to the stack



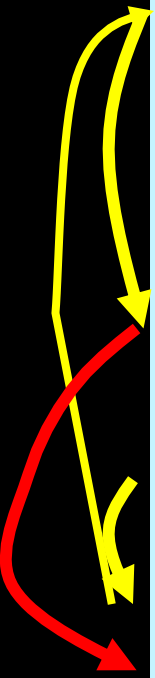
Popping Values off the stack



Return value/address

```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
}
```

```
int x, r;  
x = 42;  
r = sign(x);  
res = r + 1;  
...
```



How do we pass the **return value** to the caller?

Answer: let's use the **stack**.

Where do we keep the **return address**?

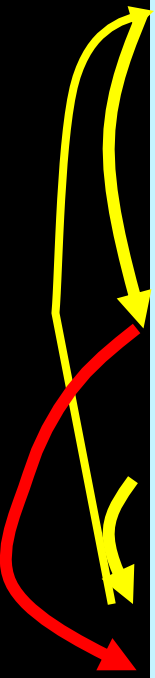
Answer: let's use **\$ra** register.

To return: **jr \$ra**

This is a design choice, NOT the only way to do it

The whole story: “when **Caller** calls **Callee**”

```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
}  
  
int x, r;  
x = 42;  
r = sign(x);  
res = r + 1;  
...
```



1. **Caller** pushes arguments to the stack
2. **Caller** stores return address to **\$ra**
3. **Callee** invoked, pop arguments from stack
4. **Callee** computes the return value
5. **Callee** pushes the return value into the stack
6. Jump to return address stored in **\$ra**
7. **Caller** pops return value from the stack.
8. Move on to next line...

Now, ready to translate the code

```
int sign (int i) {  
    if (i > 0)  
        return 1;  
    else if (i < 0)  
        return -1;  
    else  
        return 0;  
}
```

```
.text  
sign: lw $t0, 0($sp)  
      addi $sp, $sp, 4  
      bgtz $t0, gt  
      beq $t0, $zero, eq  
      addi $t1, $zero, -1  
      j end  
gt:   addi $t1, $zero, 1  
      j end  
eq:   add $t1, $zero, $zero  
end:  addi $sp, $sp, -4  
      sw $t1, 0($sp)  
      jr $ra
```

1. Callee invoked, pop arguments from stack
2. Callee computes the return value
3. Callee pushes the return value into the stack
4. Jump to return address stored in \$ra
5. Caller get return value from the stack.

Code with comments

```
.text
sign: lw $t0, 0($sp)      # pop arg i from
      addi $sp, $sp, 4    # the stack

      bgtz $t0, gt        # if ( i > 0)
      beq $t0, $zero, eq  # if ( i == 0)
      addi $t1, $zero, -1 # i < 0, return value = -1
      j end              # jump to return
gt:   addi $t1, $zero, 1  # i > 0, return value = 1
      j end              # jump to return
eq:   add $t1, $zero, $zero # i == 0, return value = 0
end:  addi $sp, $sp, -4   # push return value to
      sw $t1, 0($sp)     # the stack
      jr $ra             # return
```

Note

In Lab 10, you will implement a different convention, so don't just imitate the code in the slides for the Lab.

Insights

When we make multiple levels of function calls, the **return address** also need to be stored on stack, since the deeper level function call will overwrite the **\$ra** registers. You will experience this in Lab 10.

Before calling a function all temporary register values need to be pushed to the stack, too. After returning from the called function, you restored the register values from the stack and continue using them. This also cost time.

```
int foo() {
    int i, j;
    i=5
    j=6+i;
    # save temps to stack
    bar();
    # restore from stack
    i++;
    printf("%d %d", i, j);
}
```

Insights

What we did is based on one **function call convention** that we defined, there could be other conventions.

Function calls don't happen for free, it involves manipulating the values of several registers, and accessing memory.

All of these have performance implications.

Why "**inline functions**" are faster? Because the the callee assembly code is **inline** with the the caller code (callee code is copied to everywhere its called, rather than at a different location), so no need to jump, i.e., no stack and \$ra manipulations needed.

Now you really understand when to use inline, and when not to.

Insights

```
# NAIVE
L = [.....]
counter = 0
for x in L:
    if x % 2 == 1:
        counter +=1
return counter
```

```
# BETTER
L = [.....]
counter = 0
for x in L:
    counter += (x % 2)
return counter
```

Insights

- Branching or jumping is in general less efficient than linear execution.
- Modern processors do **prefetching**, i.e., while you are executing a line of instruction, the next line is already loaded from memory before you're sure that's the next line to execute.
 - This is made possible by using a **pipelined** architecture which enables **parallelism**
 - It's a speed boost, but it'd be wasted if you branch to a line different from the next line.
- More advanced modern processors have **branch predictors**, which try to guess which way the branch goes before it's known for sure.
 - Most modern branch predictors have > 90% accuracy.
 - Pro performance tip: know how your branch predictor works, and write your code in such a way that the branch prediction is more likely to be correct.

People who are really serious about software should make their own hardware.

-- Steve Jobs quoting Alan Kay

```
i = 0
while i < 10000:
    print(i)
    i += 1
```

```
# Better performance
i = 0
while i < 10000:
    print(i)
    i += 1
    print(i)
    i += 1
    print(i)
    i += 1
    print(i)
    i += 1
    print(i)
    i += 1
    print(i)
    i += 1
    print(i)
    i += 1
    print(i)
    i += 1
    print(i)
    i += 1
    print(i)
    i += 1
    print(i)
    i += 1
```


Practice for home: String function

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

Translated string program

```
strcpy:      lw      $a0, 0($sp)           # pop x address
             addi   $sp, $sp, 4         # off the stack

             lw      $a1, 0($sp)           # pop y address
             addi   $sp, $sp, 4         # off the stack

L1:          add    $s0, $zero, $zero      # $s0 = offset i
             add    $t1, $s0, $a0        # $t1 = x + i
             lb     $t2, 0($t1)         # $t2 = x[i]
             add    $t3, $s0, $a1        # $t3 = y + i
             sb     $t2, 0($t3)         # y[i] = $t2
             beq   $t2, $zero, L2        # y[i] = '\0'?
             addi  $s0, $s0, 1          # i++

L2:          j      L1                  # loop
             addi  $sp, $sp, -4         # push i onto
             sw    $s0, 0($sp)         # top of stack
             jr    $ra                  # return
```

initialization

main algorithm

end