# CSC258 Winter 2016
# Lecture 10

# Announcements

Midterm requests all processed. You can pick them up after the lecture.

Final exam schedule: Monday April 11, 9am~12pm, in IB110


# Quiz Time!

## Tear off the reference sheet.

## This quiz has 6 marks

# Question 1

Complete the following assembly code, which **multiplies** the values in $t1 and $t2, and stores the result in **$t3**. Assume that the result is small enough to be represented by 32 bits.

```
mult $t1, $t2

___________________
```

# Question 2

`blt` is a pseudo-instruction, the following line

```
blt $t1, $t2, Label
```

is equivalent to the following two lines combined. Complete them

```
___ $t3, $t1, $t2
bne $t3, ____, Label
```

# Question 3

Translate the C program on the right by completing the following assembly code.

```
if (a > b && c > b) {
    b++;
}
else {
    b--;
}
c = a + b;
```

```
#  $t1 = a, $t2 = b, $t3 = c
IF:
      ble $t1, $t2, ________
      bge $t2, $t3, ________
THEN:
      addi $t2, $t2, 1

      ____________________
ELSE:
      addi $t2, $t2, -1
END:
      add $t3, $t1, $t2
```

# Solutions

# Question 1

Complete the following assembly code, which **multiplies** the values in $t1 and $t2, and stores the result in **$t3**. Assume that the result is small enough to be represented by 32 bits.

```
mult $t1, $t2
mflo $t3
```

# Question 2

`blt` is a pseudo-instruction, the following line

```
blt $t1, $t2, Label
```

is equivalent to the following two lines, complete them

```
slt $t3, $t1, $t2
bne $t3, $zero, Label
```

Just "0" is not right

# Question 3

Translate the C program on the right into assembly by completing the following code.

```
if (a > b && c > b) {
    b++;
}
else {
    b--;
}
c = a + b;
```

```
# $t1 = a, $t2 = b, $t3 = c
IF:
      ble $t1, $t2, ELSE
      bge $t2, $t3, ELSE
THEN:
      addi $t2, $t2, 1
      j END
ELSE:
      addi $t2, $t2, -1
END:
      add $t3, $t1, $t2
```

# Loops

are easy once you understand how if-else (branch) works

# Loops in MIPS (while loop)

- Example of a simple loop, in assembly:

```
#   $t0 = i, $t1 = n
main:    add $t0, $zero, $zero    # i = 0
         addi $t1, $zero, 100     # n = 100
START:   beq $t0, $t1, END        # if i == n, END
         addi $t0, $t0, 1         # i++
         j START
END:
```

- …which is the same as saying (in C):

```
int i = 0;
while (i != 100) {
   i++;
}
```

# For loop

```
for ( <init> ; <cond> ; <update> ) {
    <for body>
}
```

- For loops (such as above) are usually implemented with the following structure:

```
main:    <init>
START:   if (!<cond>) branch to END
          <for-body>
UPDATE:   <update>
          jump to START
END:
```

# Exercise:

```
j = 0
for ( _____ ; _____ ; _____ )
{
     j = j + i;
}
```

```
#  $t0 = i, $t1 = j
main:    add $t1, $zero, $zero        # set j = 0
         add $t0, $zero, $zero        # set i = 0
         addi $t9, $zero, 100         # set $t9 to 100
START:   beq $t0, $t9, EXIT           # branch if i==100
         add $t1, $t1, $t0            # j = j + i
UPDATE:  addi $t0, $t0, 1             # i++
         j START
EXIT:
```

# Answer

```
j = 0
for ( i=0 ; i!=100 ; i++ )
{
      j = j + i;
}
```

- This translates to:

```
#  $t0 = i, $t1 = j
main:    add $t1, $zero, $zero        # set j = 0
         add $t0, $zero, $zero        # set i = 0
         addi $t9, $zero, 100         # set $t9 to 100
START:   beq $t0, $t9, EXIT           # branch if i==100
         add $t1, $t1, $t0            # j = j + i
UPDATE: addi $t0, $t0, 1              # i++
         j START
EXIT:
```

- `while` loops are the same, without the initialization and update sections.

# Another exercise

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

```
int fib(void) {
    int n = 10;
    int f1 = 1, f2 = -1;

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

# Assembly code example

- Fibonacci sequence in assembly code:

```
int fib(void) {
   int n = 10;
   int f1 = 1, f2 = -1;

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

```
# fib.s
# register usage: $t3=n, $t4=f1, $t5=f2
# RES refers to memory address of result
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:  sw $t4, RES             # store result
```

# Making an assembly program

- Assembly language programs typically have structure similar to simple Python or C programs:
  - They set aside registers to store data.
  - They have sections of instructions that manipulate this data.
- It is always good to decide at the beginning which registers will be used for what purpose

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:

Load or store

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

Register storing address of data value in memory

Local data register

Offset from memory address

# 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)
```

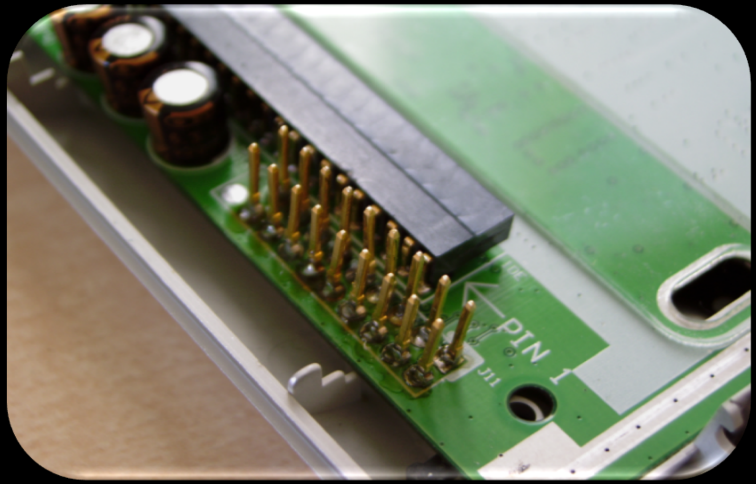
Load a half-word (2 bytes) starting from MEM($s + 12), sign-extend it to 4 bytes, and store in `$t0`

```
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 offset value 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.




| 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 $ao and $vo

`$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 doubleread |
| **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 syscal 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 beginning of 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];
```

| | | | | | |
|---|---|---|---|---|---|
| A[0] | A[1] | A[2] | A[3] | A[4] | ... |

Offset = 4 x 4 bytes = 16 bytes

Address of A[4] = Address of A[0] + 16

```
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 ( addr(A)+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` and `sb` lines indicate that values in `$t1` are being stored at `$t0`, `$t0+4` and `$t0+5`.
  - Each previous line sets the value of `$t1` to store.
- Therefore, this code stores the values `5`, `‘B’ (ascii 66)` and `12` into the struct at location `a1`.

```
            .data
a1:         .space     9

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

# Example: A struct program

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

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

```
        .data
a1:     .space  9

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

# Struct program with comments

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

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

```
.data
a1:     .space   9                  #  declare 9 bytes
                                    #  of storage to hold
                                    #  struct of 2 ints and
                                    #  1 char

.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, 'B'    #  $t1 = 'B', i.e., 66
        sb       $t1, 4($t0)        #  second struct
                                    #  element set to 'B'
        addi     $t1, $zero, 12     #  $t1 = 12
        sw       $t1, 5($t0)        #  third struct
                                    #  element set to 12
```

# Function calls

# Another example:

A function!

Function arguments!

Return!

```
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 = r + 1;
…
```

# Function arguments

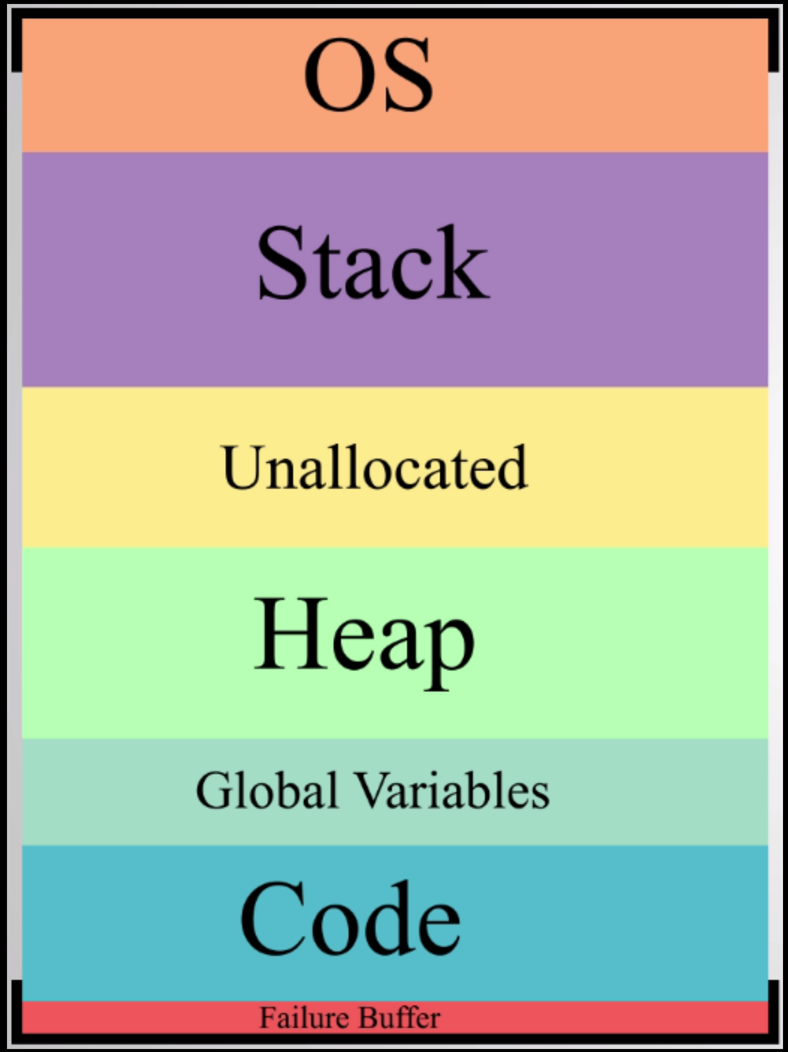
```
int sign (int i) {
    if (i > 0)
        return 1;
    else if (i < 0)
        return -1;
    else
        return 0;
}

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

# Memory model: a quick look




High address

Stack grows this way (going low)

If they collide

stackoverflow

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);
r = res + 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

Of course, 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 complier 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**

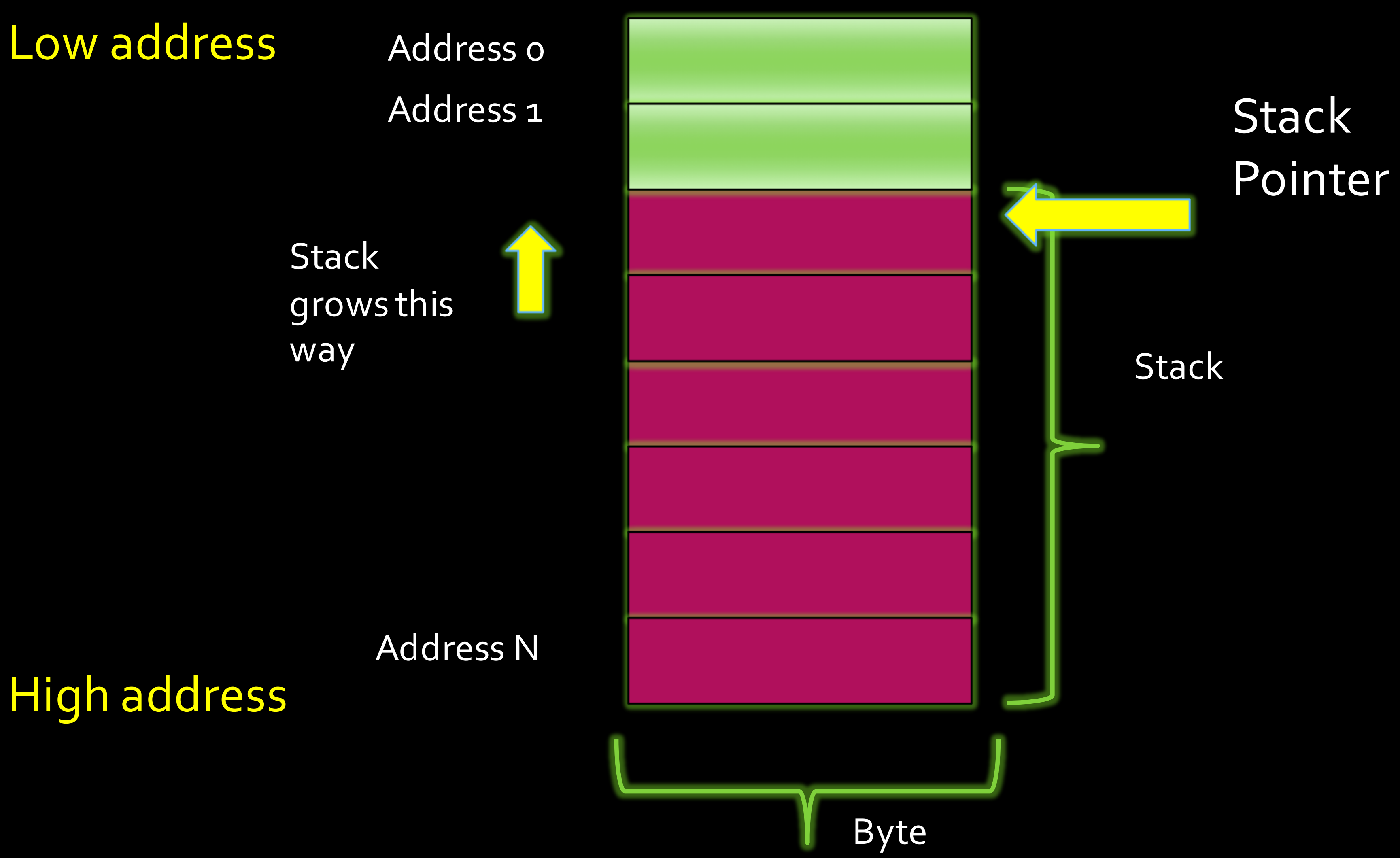
## **PUSH** value in $t0 into stack

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

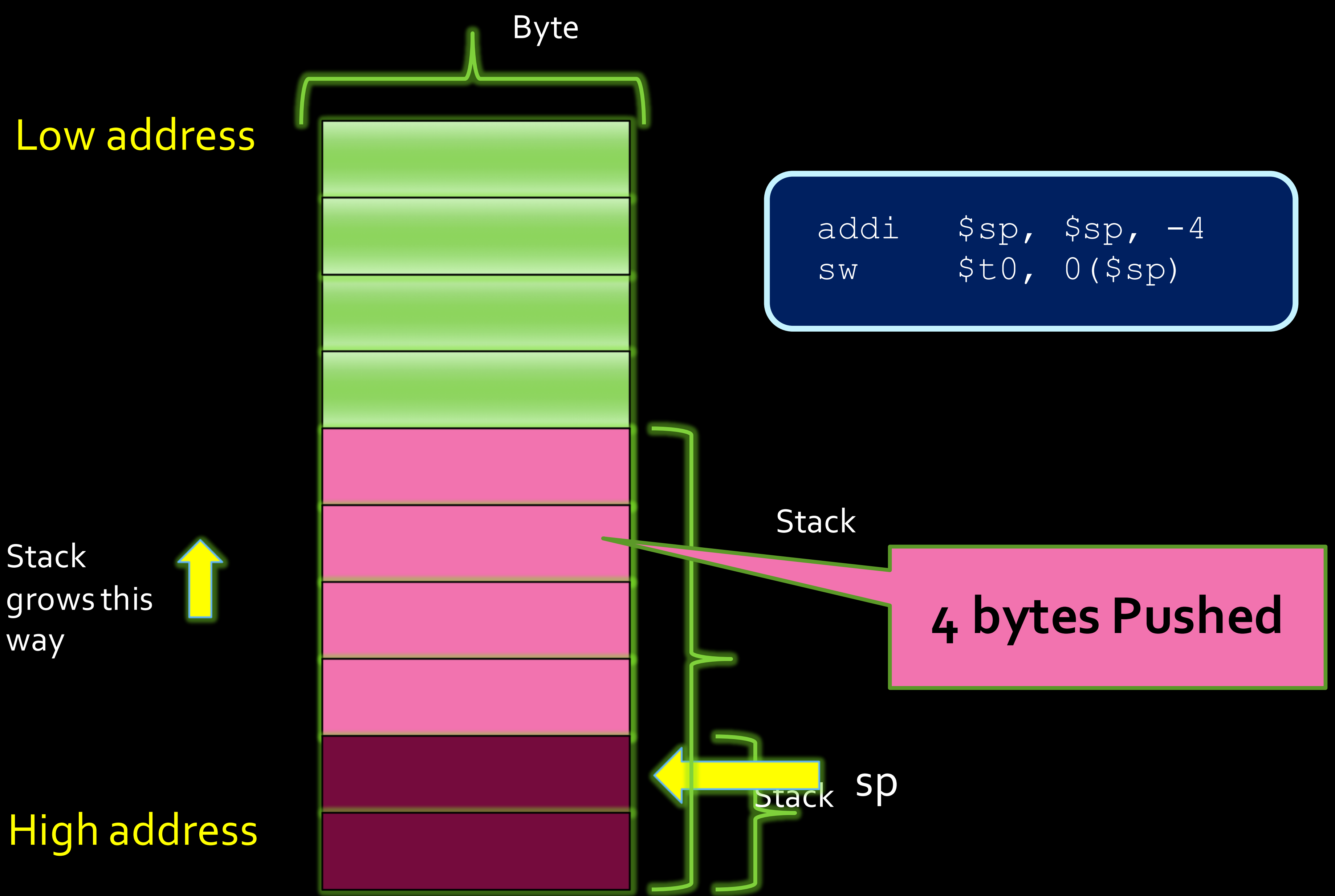
## **POP** a value from stack and store in $t0

```
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 Pointer
Stack grows this way
Stack
Address N
High address
Byte

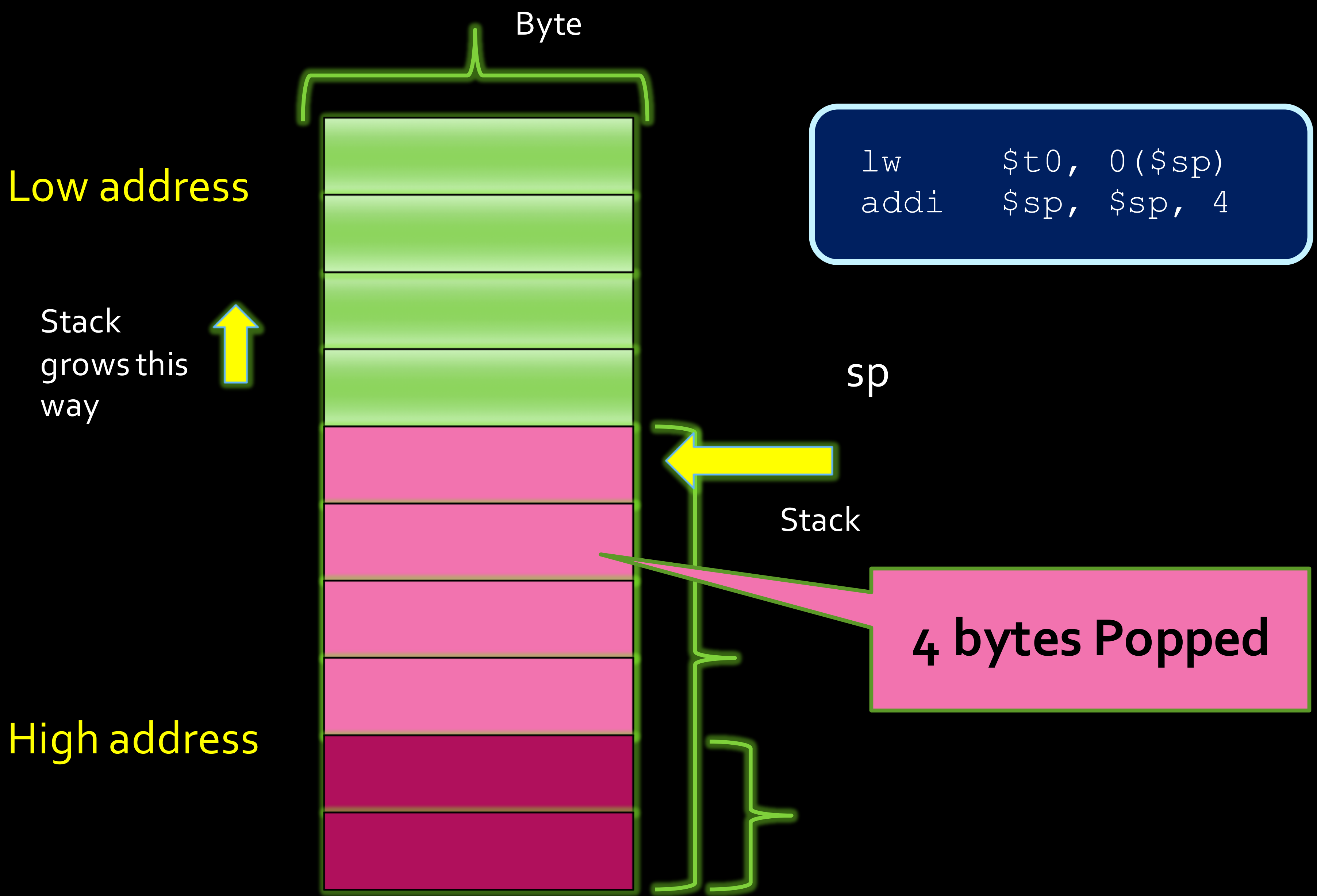
# Pushing Values to the stack




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

# Popping Values off the stack

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



# Return 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);
r = res + 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);
r = res + 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 addressed 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 addressed 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
```

# Takeaway

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.

# 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
   initialization  lw    $a1, 0($sp)          # pop y address
                   addi  $sp, $sp, 4          #  off the stack
                   add   $s0, $zero, $zero    # $s0 = offset i
L1:                add   $t1, $s0, $a0        # $t1 = x + i
                   lb    $t2, 0($t1)          # $t2 = x[i]
                   add   $t3, $s0, $a1        # $t3 = y + i
  main algorithm   sb    $t2, 0($t3)          # y[i] = $t2
                   beq   $t2, $zero, L2       # y[i] = '/0'?
                   addi  $s0, $s0, 1          # i++
                   j     L1                   # loop
L2:                addi  $sp, $sp, -4         # push i onto
              end  sw    $s0, 0($sp)          #  top of stack
                   jr    $ra                  # return
```

# Next one

```
int factorial (int n) {
   if (n == 0)
     return 1;
   else
     return n * factorial(n-1);
}
```

Recursion!