CSC258 Winter 2016
Lecture 11

Announcements

QUIZ FINALS

Tear off the reference sheet

Question 1

Complete the assembly translation of the C code.

```
int i;
int j = 258;
for (i = 42; j > i; i = i*2) {
    j++;
}
i = j;
```

```
# t1 = i, t2 = j
START:
    li $t2, 258
    li $t1, 42
LOOP:
    ble $t2, $t1, END
    addi $t2, $t2, 1

    ________________________

    ________________________
END:
    move $t1, $t2
```

Question 2

What is the content of array after executing the following code?

```
.data
array:        .word     2, 5, 8
.text
main:
        la $s0, array
        lw $t0, 4($s0)
        addi $t0, $t0, 1
        sw $t0, 4($s0)
        lw $t0, 0($s0)
        addi $t0, $t0, 1
        addi $s0, $s0, 8
        sw $t0, 0($s0)
```

Question 3

Complete the assembly code (3 numbers to be filled) so that it is equivalent to the C code with struct.

```
struct foo {
    char A;          # 1-byte
    short int B;     # 2-byte
    int C;           # 4-byte
};

struct foo x;
x.A = 55;
x.B = 66;
x.C = 77;
```

```
            .data
s1:         .space      ____

            .text
main:       la          $t0, s1
            li          $t1, 55
            sb          $t1, 0($t0)
            li          $t1, 66
            sh          $t1, ____($t0)
            li          $t1, 77
            sw          $t1, ____($t0)
```

Solution

Question 1

Complete the assembly translation of the C code.

```
int i;
int j = 258;
for (i = 42; j > i; i = i*2) {
    j++;
}
i = j;
```

```
# t1 = i, t2 = j
START:
    li $t2, 258
    li $t1, 42
LOOP:
    ble $t2, $t1, END
    addi $t2, $t2, 1
    sll $t1, $t1, 1
    j LOOP
END:
    move $t1, $t2
```

Question 2

What is the content of array after executing the following code?

```
.data
array:          .word     2, 5, 8
.text
main:
        la $s0, array       # load addr of A
        lw $t0, 4($s0)      # load A[1]: 5
        addi $t0, $t0, 1    # 5 + 1 = 6
        sw $t0, 4($s0)      # store A[1] = 6
        lw $t0, 0($s0)      # load A[0]: 2
        addi $t0, $t0, 1    # 2 + 1 = 3
        addi $s0, $s0, 8    # $s0 changed to A[2] addr
        sw $t0, 0($s0)      # store A[2] = 3
```

2, 6, 3

Question 3

Complete the assembly code so that it is equivalent to the C code with struct.

```
struct foo {
    char A;         # 1-byte
    short int B;    # 2-byte
    int C;          # 4-byte
};

struct foo x;
x.A = 55;
x.B = 66;
x.C = 77;
```

```
          .data
s1:       .space    7

          .text
main:     la        $t0, s1
          li        $t1, 55
          sb        $t1, 0($t0)
          li        $t1, 66
          sh        $t1, 1($t0)
          li        $t1, 77
          sw        $t1, 3($t0)
```

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?

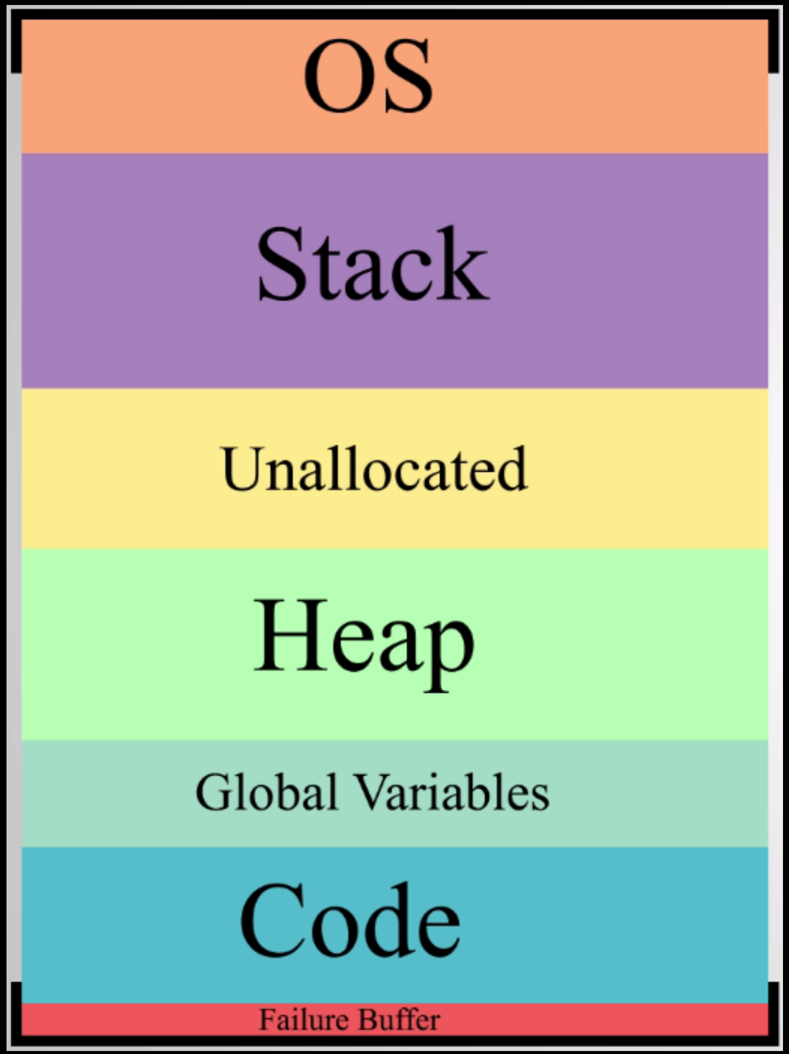
They are stored at a certain location in the memory,
which is call the **stack**.

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

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

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

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**

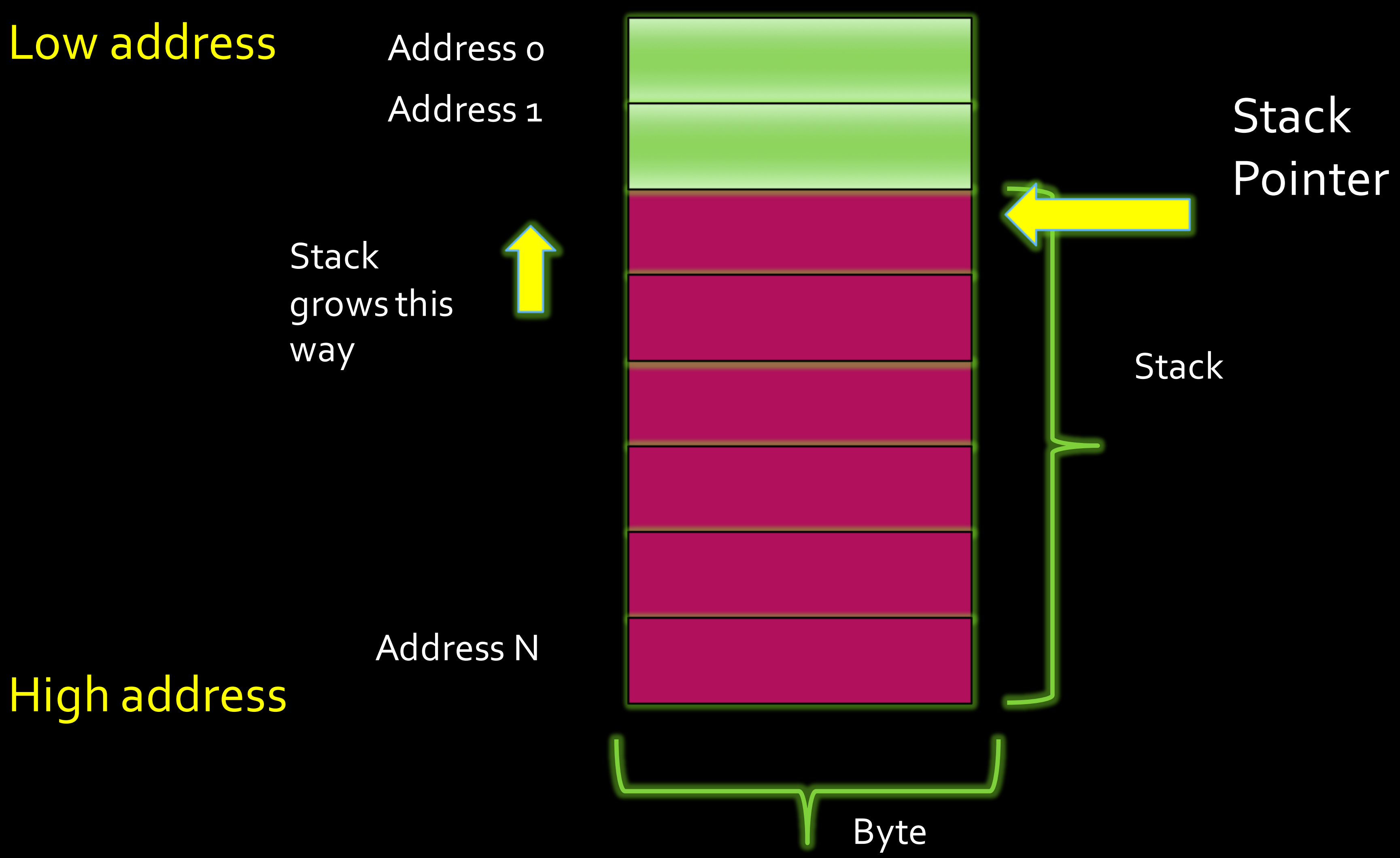
PUSH value in $to into stack

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

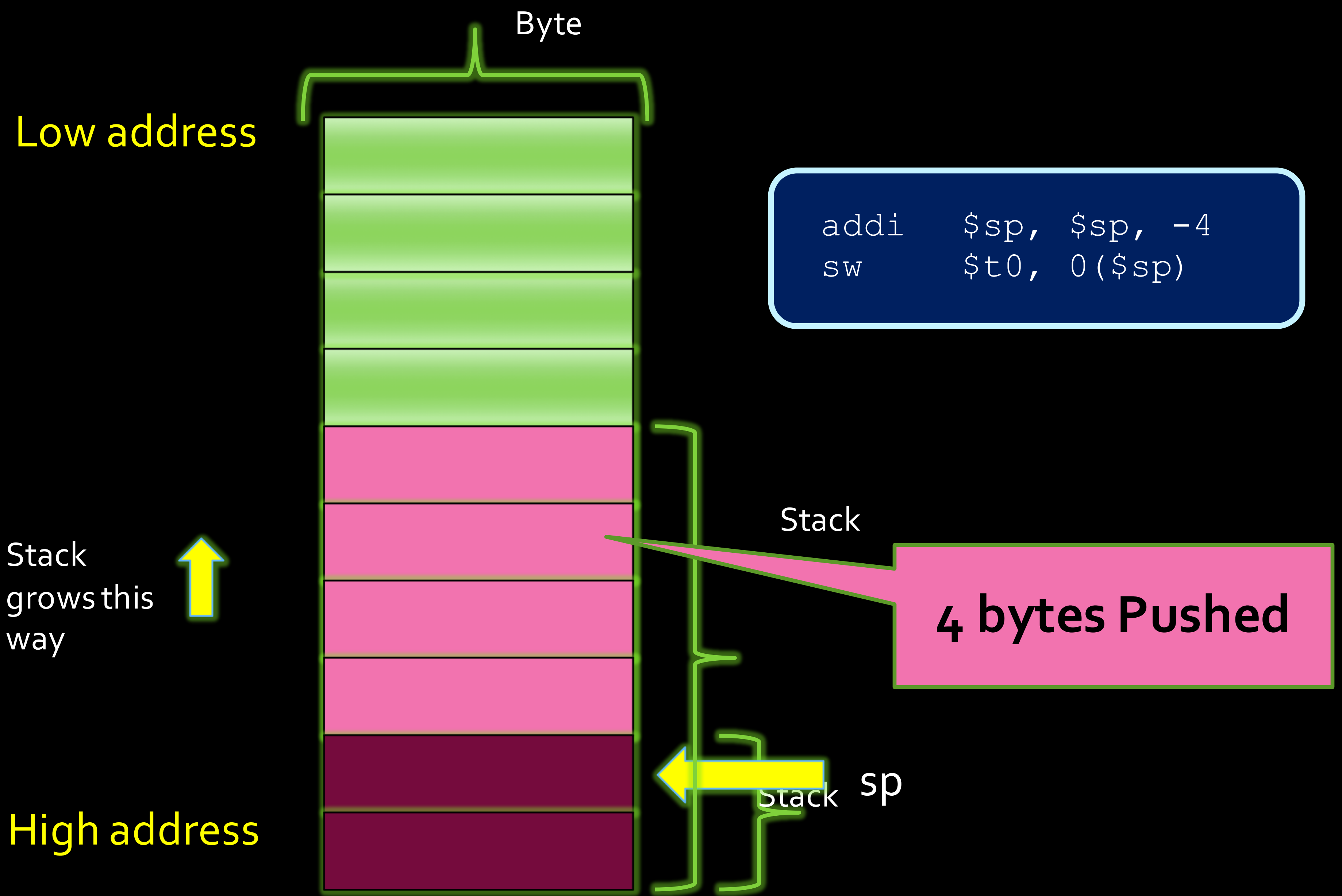
POP a value from stack and store in $to

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

The Stack

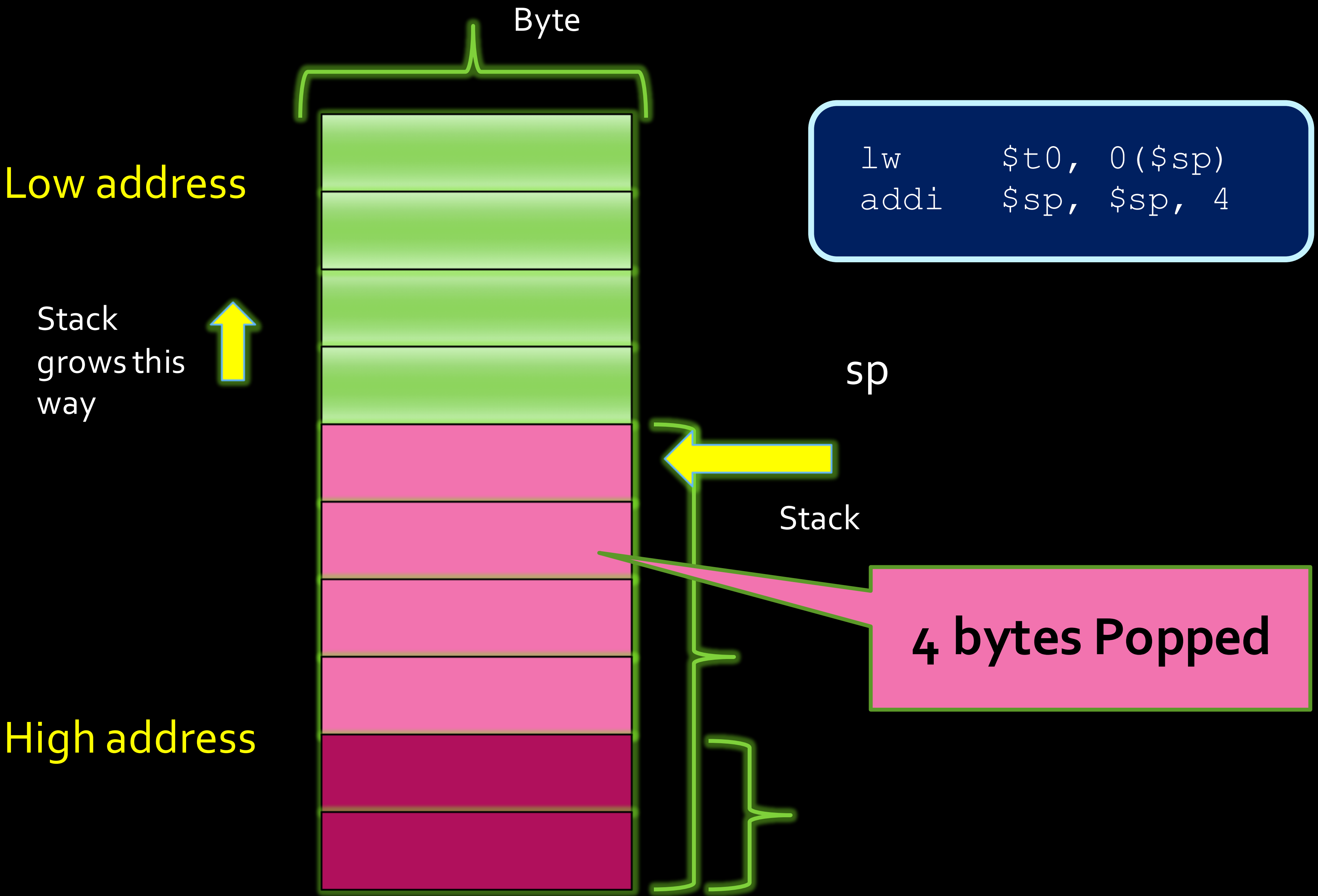

Low address
Address 0
Address 1
Stack Pointer
Stack grows this way
Stack
Address N
High address
Byte

Pushing Values to the stack



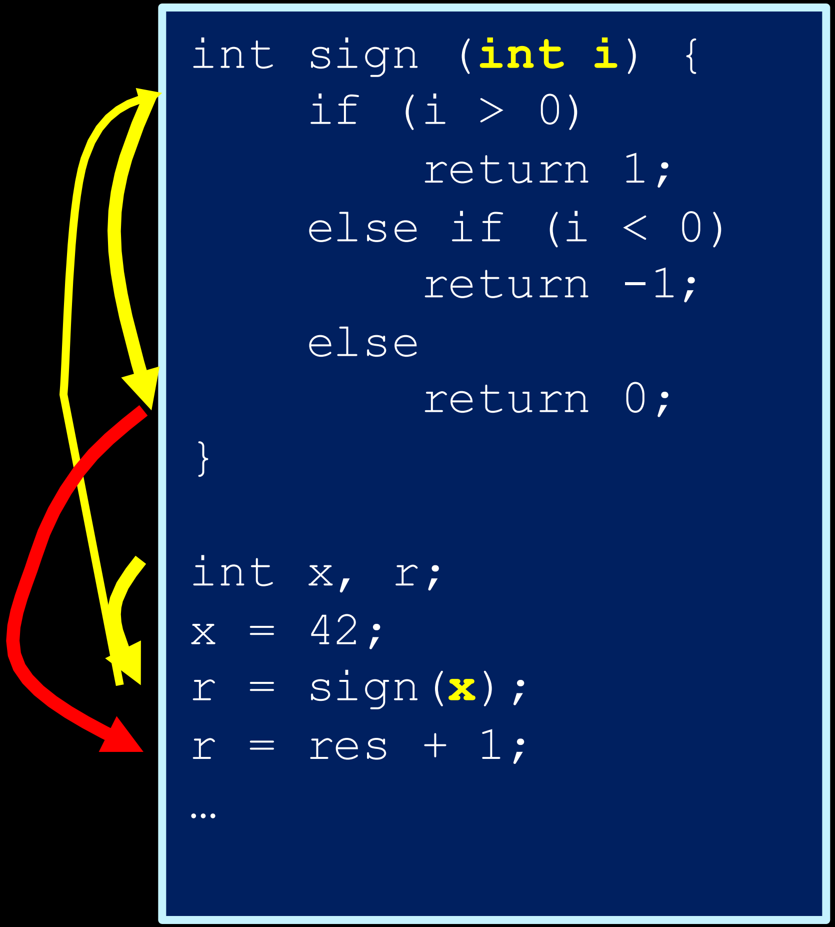
```
addi   $sp, $sp, -4
sw     $t0, 0($sp)
```

Popping Values off the stack



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

Return value/address



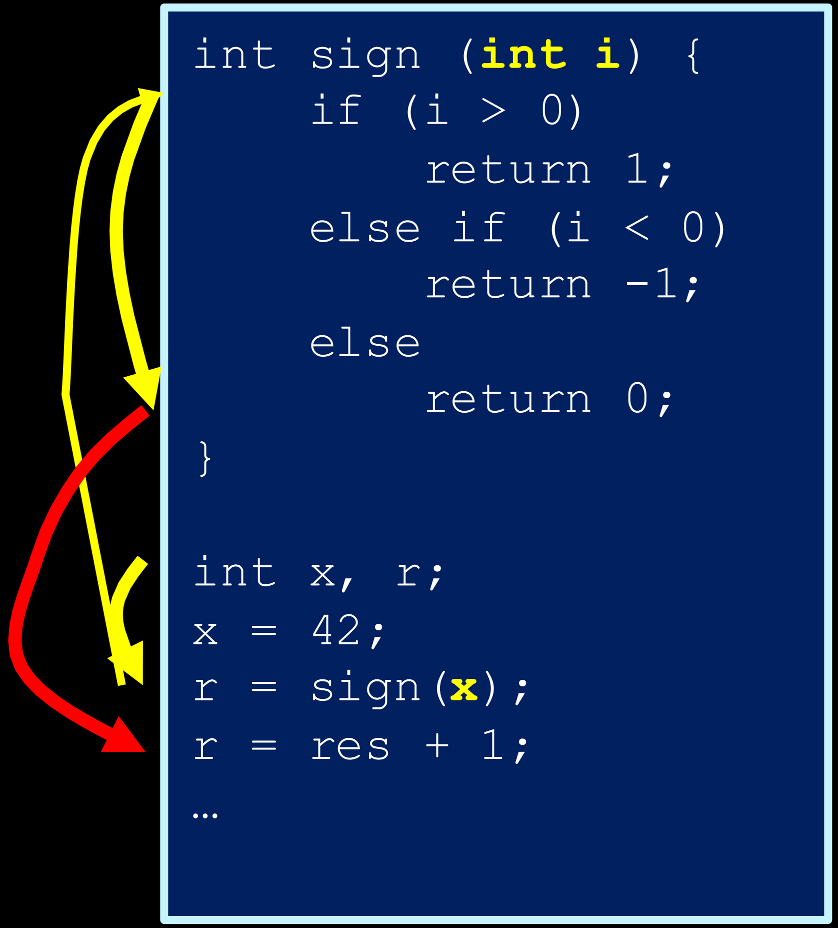
```
int sign (int i) {
    if (i > 0)
        return 1;
    else if (i < 0)
        return -1;
    else
        return 0;
}

int x, r;
x = 42;
r = sign(x);
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
```

Note

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

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.

More takeaway

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.

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

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]
main algorithm      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++
                    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!

Recursion in Assembly

what recursion really is in hardware

I WANNA UNDERSTAND RECURSION
SURE, BUT YOU HAVE TO UNDERSTAND IT FIRST
imgflip.com

Example: `factorial(int n)`

- Basic pseudocode for recursive factorial:

```
Base Case (n == 0)
    return 1
Get factorial(n-1)
    Store result in "product"
Multiply product by n
    Store in "result"
Return result
```

```
factorial(3)
  p = 3 * factorial(2)
    factorial(2)
      p = 2 * factorial(1)
        factorial(1)
          p = 1*factorial(0)
            factorial (0)
              p = 1 # Base!
              return p
          return p
      return p
  return p
```

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

Before writing assembly, we need to know explicitly where to store values

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

Need to store …

- the value of n
- the value of n – 1
- the value factorial(n-1)
- the return value: 1 or n*factorial(n-1)

Design decision #1: store values in registers

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

Does it work?

- store n in $t0
- store n-1 in $t1
- store factorial(n-1) in $t2
- store return value in $t3

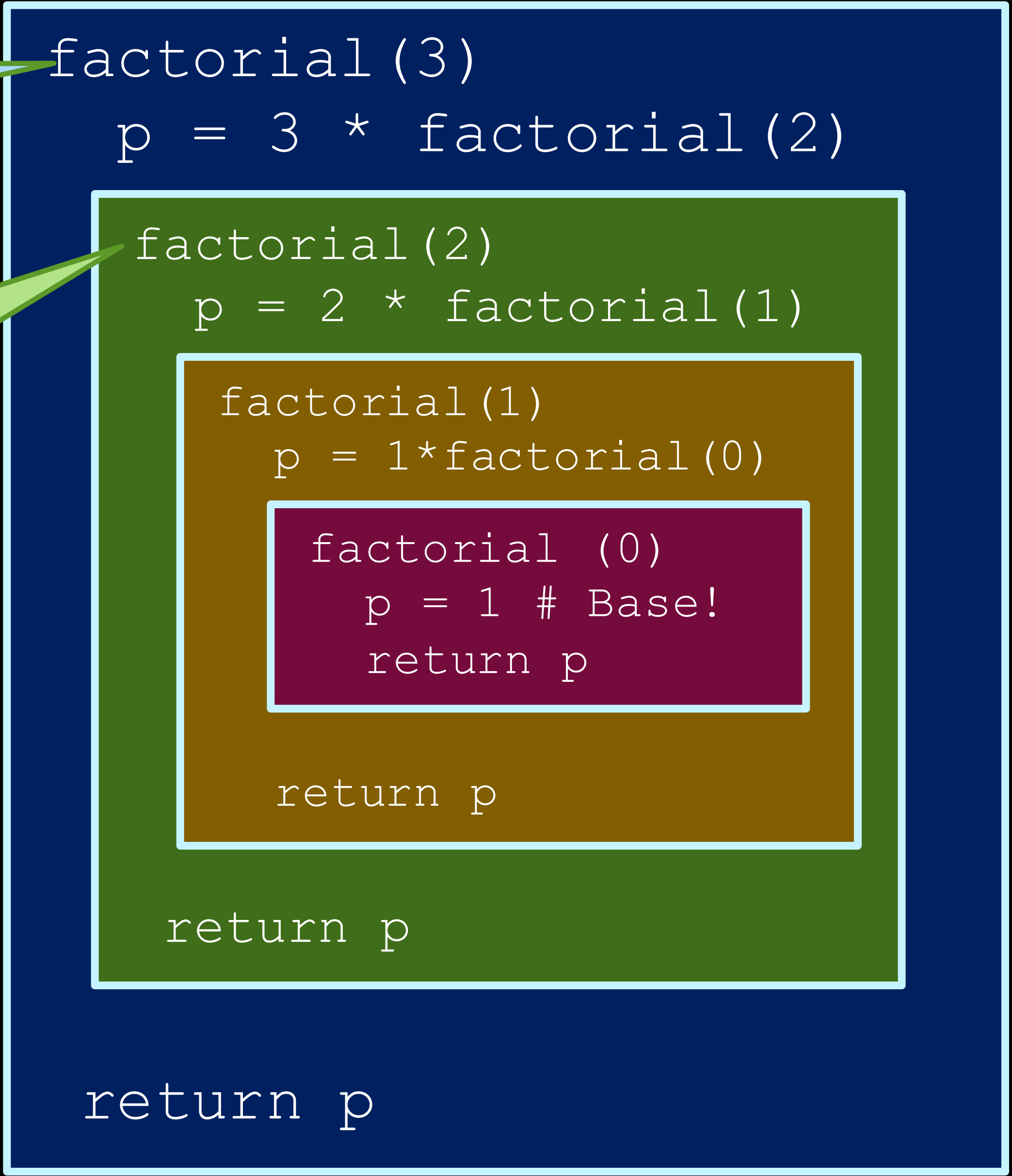
No, it doesn't work.

Store **n=3** in $t0

Store **n=2** in $t0, the stored 3 is overwritten, lost!

Same problem for $t1, t2, t3

- store n in $t0
- store n-1 in $t1
- store factorial(n-1) in $t2
- store return value in $t3



```
factorial(3)
  p = 3 * factorial(2)
    factorial(2)
      p = 2 * factorial(1)
        factorial(1)
          p = 1*factorial(0)
            factorial (0)
              p = 1 # Base!
              return p
          return p
      return p
  return p
```

A register is like a laundry basket -- you put your stuff there, but when you call another function (person), that person will use the same basket and take / mess up your stuff.

And yes, the other person will guarantee to use the **same** basket because … the other person is **YOU**! (because recursion)

So the correct design decision is to use __Stack___ .

Each recursive call has its own space for storing the values

Stores **n=2** for factorial (2)

Stores **n=3** for factorial (3)

Stacking Laundry Baskets are ideal for sorting laundry in small spaces.

Two useful things about stack

1. It has a lot of space
2. Its LIFO order (last in first out) is suitable for implementing recursions

LIFO order & recursive calls

Note: Everybody is getting the **correct** basket because of LIFO!

```
factorial(2)
  p = 2 * factorial(1)
   factorial(1)
     p = 1*factorial(0)
      factorial (0)
        p = 1 # Base!
        return p

     return p

  return p
```

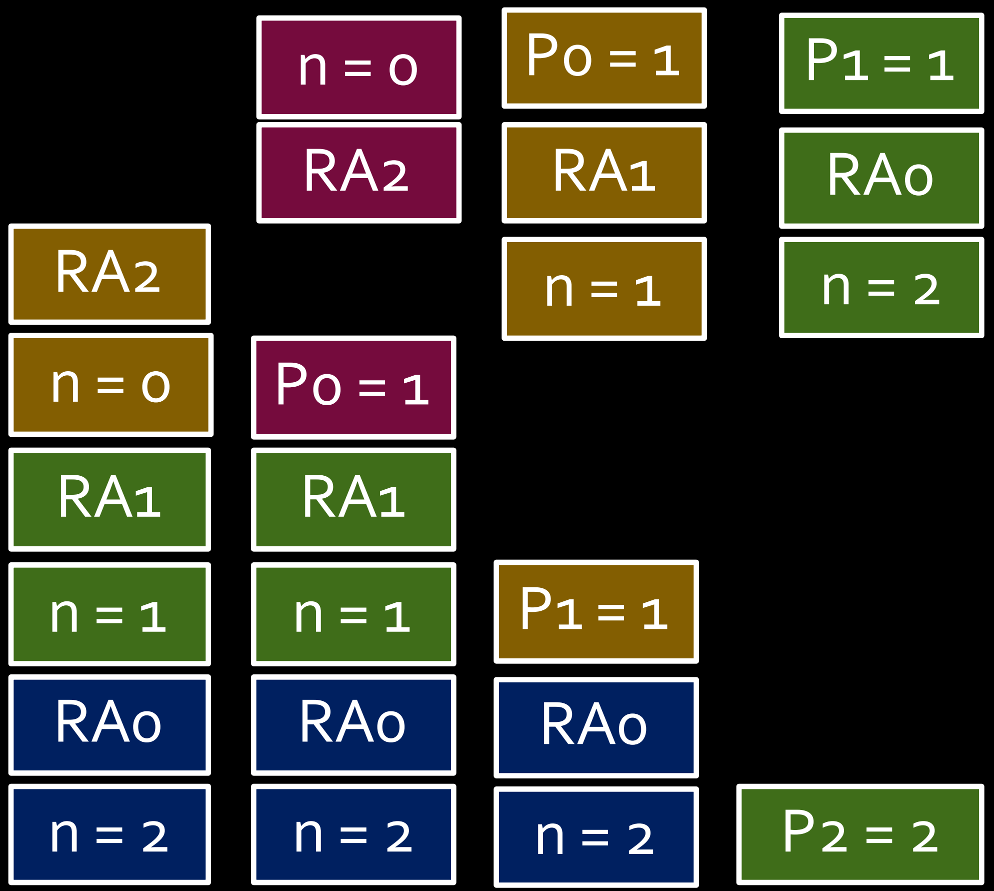
Stacking Laundry Baskets are ideal for sorting laundry in small spaces.

Design decisions made, now let’s actually write the assembly code

LIFO order & recursive calls

```
int x = 2;
int y = factorial(x)
print(y) # RA0
```

```
factorial(n=2)
  r = factorial(1)

  factorial(n=1)
    r = factorial(0)

    factorial(n=0)
      p = 1 # Base!
      return p #P0

    p = n * r;  # RA2
  return p #P1

  p = n * r; # RA1
return p # P2
```



Actions in factorial (n)

Before making the recursive call

- pop argument n
- push argument n-1 (arg for recursive call)
- push return address (remember where to return)
- make the recursive call

After finishing the recursive call

- pop return value from recursive call
- pop return address
- compute return value
- push return value (so the upper call can get it)
- jump to return address

factorial(int n)

n → \$t0

n-1 → \$t1

fact(n-1) → \$t2

- Pop n off the stack
  - Store in \$t0
- If \$t0 == 0,
  - Push return value 1 onto stack
  - Return to calling program
- If \$t0 != 0,
  - Push \$t0 and \$ra onto stack
  - Calculate n-1
  - Push n-1 onto stack
  - Call factorial

    - ...time passes...
  - Pop the result of factorial (n-1) from stack, store in \$t2
  - Restore \$ra and \$t0 from stack
  - Multiply factorial (n-1) and n
  - Push result onto stack
  - Return to calling program