Week 9, part D: Functions and the Stack

Functions in Assembly

- Functions allow us to reuse code by creating an interface to that specifies inputs and outputs.
- Once a function finishes, control returns to the caller, optionally with return value.
- MIPS assembly does not really understand functions…
- … but the MIPS designers do.
  - They intentionally designed functionality that makes implementing functions easier.

A Simple C Function

function name

arguments

return value

implementation

```
int sign (int n) {
    if (n > 0)
        return 1;
    else if (n == 0)
        return 0;
    else
        return -1;
}

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

2. run function code

3. finish

1. call function with arguments

4. return here

How do we do this in assembly?

Implementing Functions

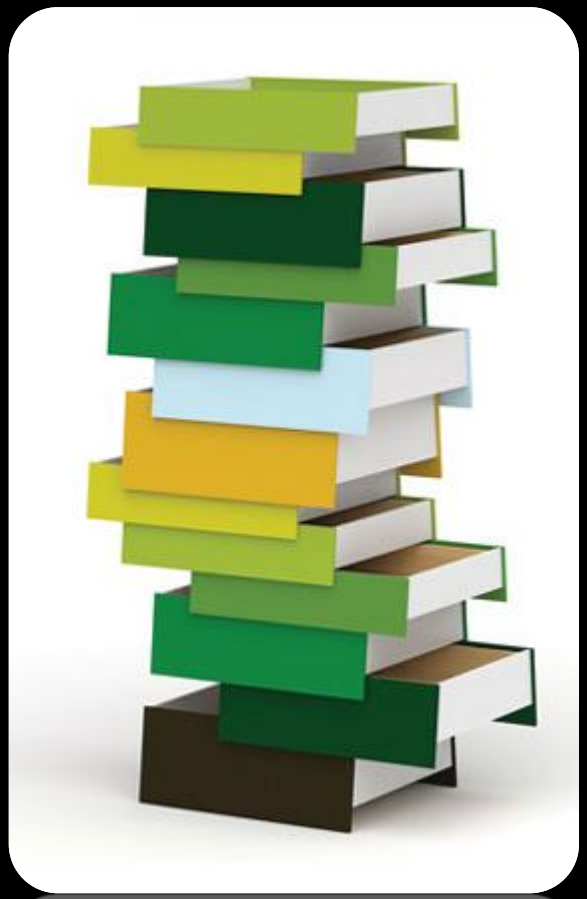
- We already know how to do some things.
- We can jump to a block of code and jump back
  - How do we know where to jump back to?
- We can implement functions that have no parameters or return value
  - Not very useful
  - How do we pass parameters and returned value?
- We'll start with the second question first.

Parameters: Option #1

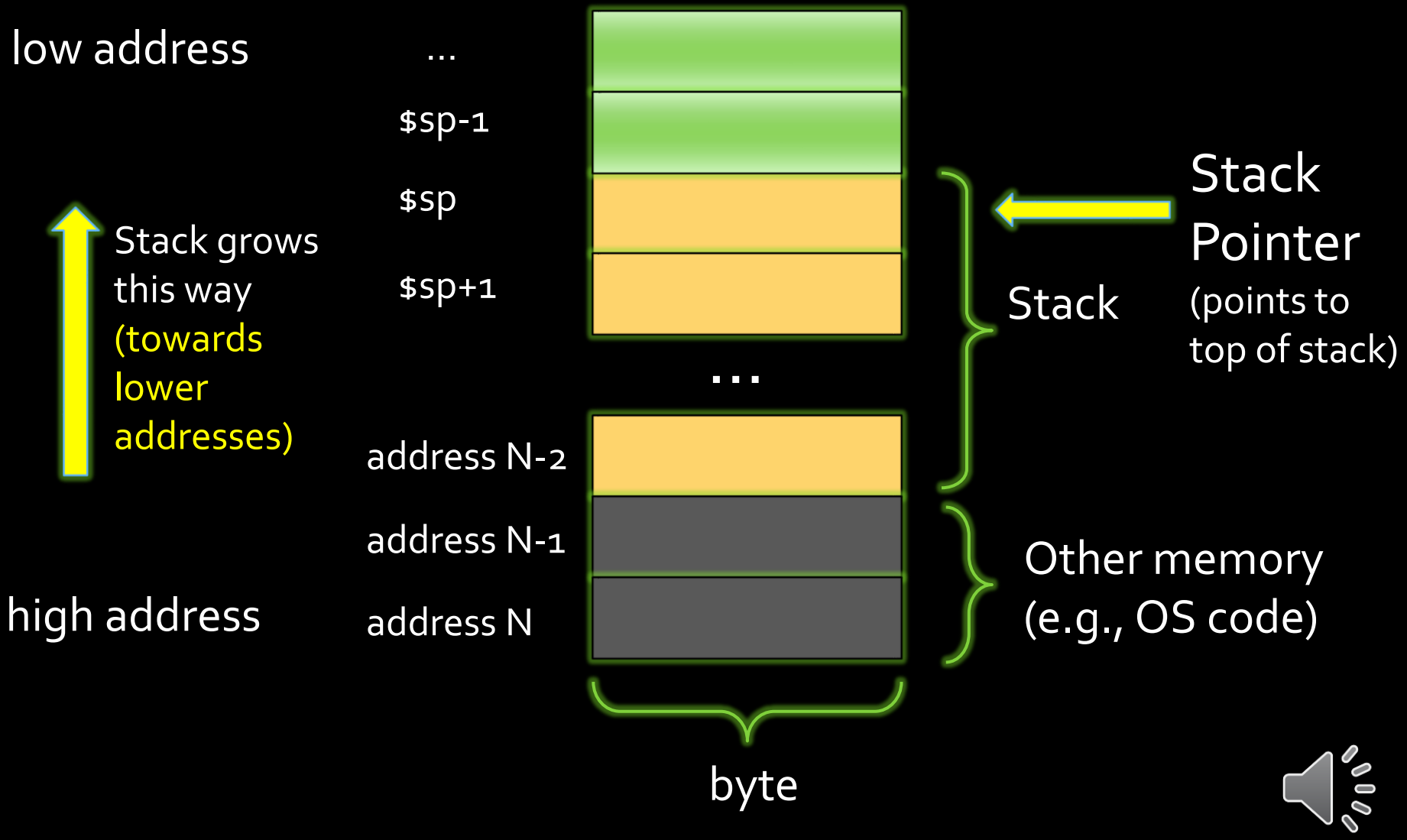
- Reserve some registers for parameters & return values
- Remember the register table?
  - Registers 2–3 (`$v0`, `$v1`): return values
  - Registers 4–7 (`$a0`-`$a3`): function arguments
- Problems?
  - What if we need more parameters?
  - What if that function calls another function?
  - Recursion?

Parameters: Option #2

- Use a stack : an area in memory set aside for this.
- $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.



The Stack, illustrated



Pushing on Stack and Popping

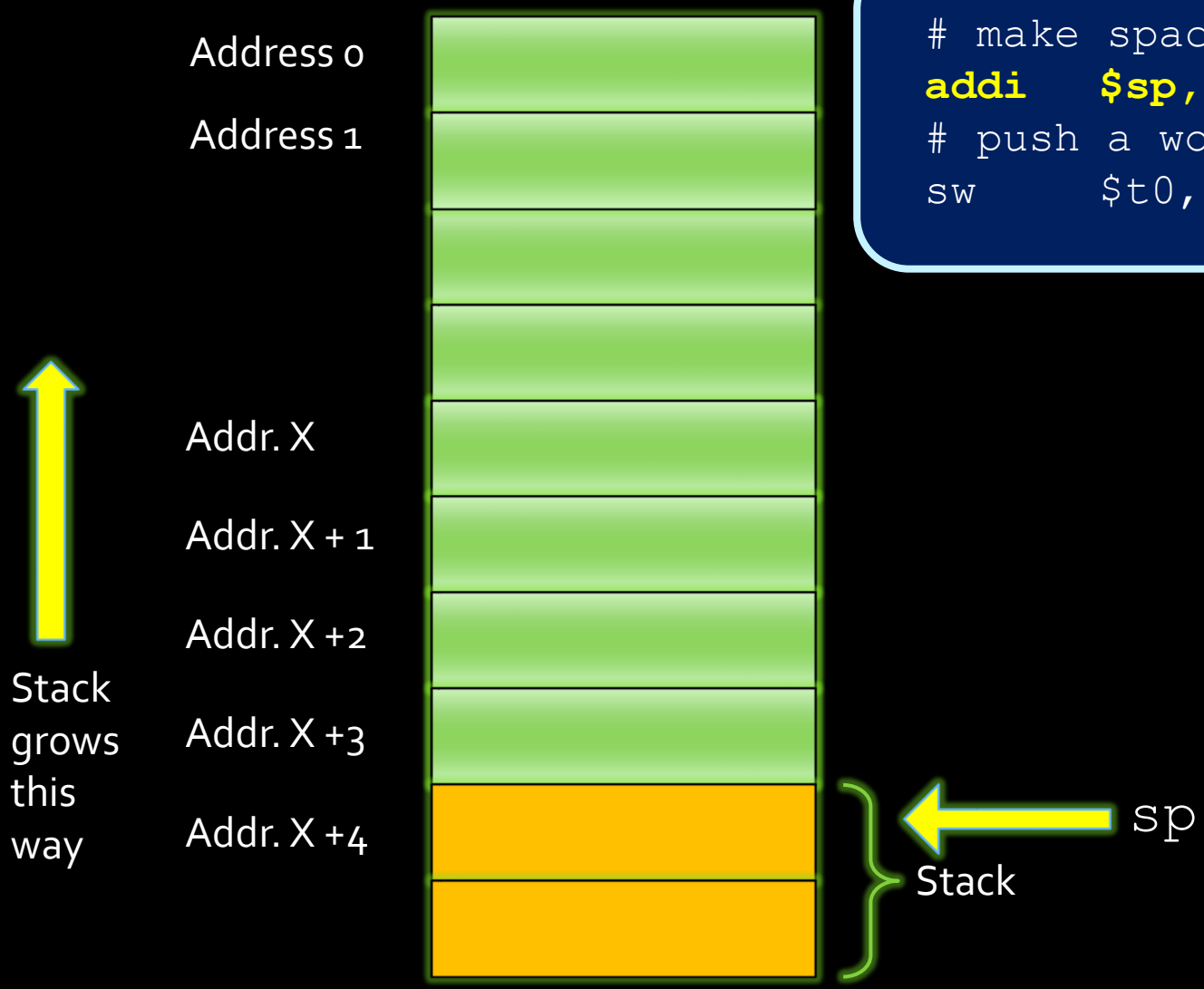
- The address of the top of the stack (stack pointer) is stored in register `$sp`
- PUSH value `$t0` onto the stack

```
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
```

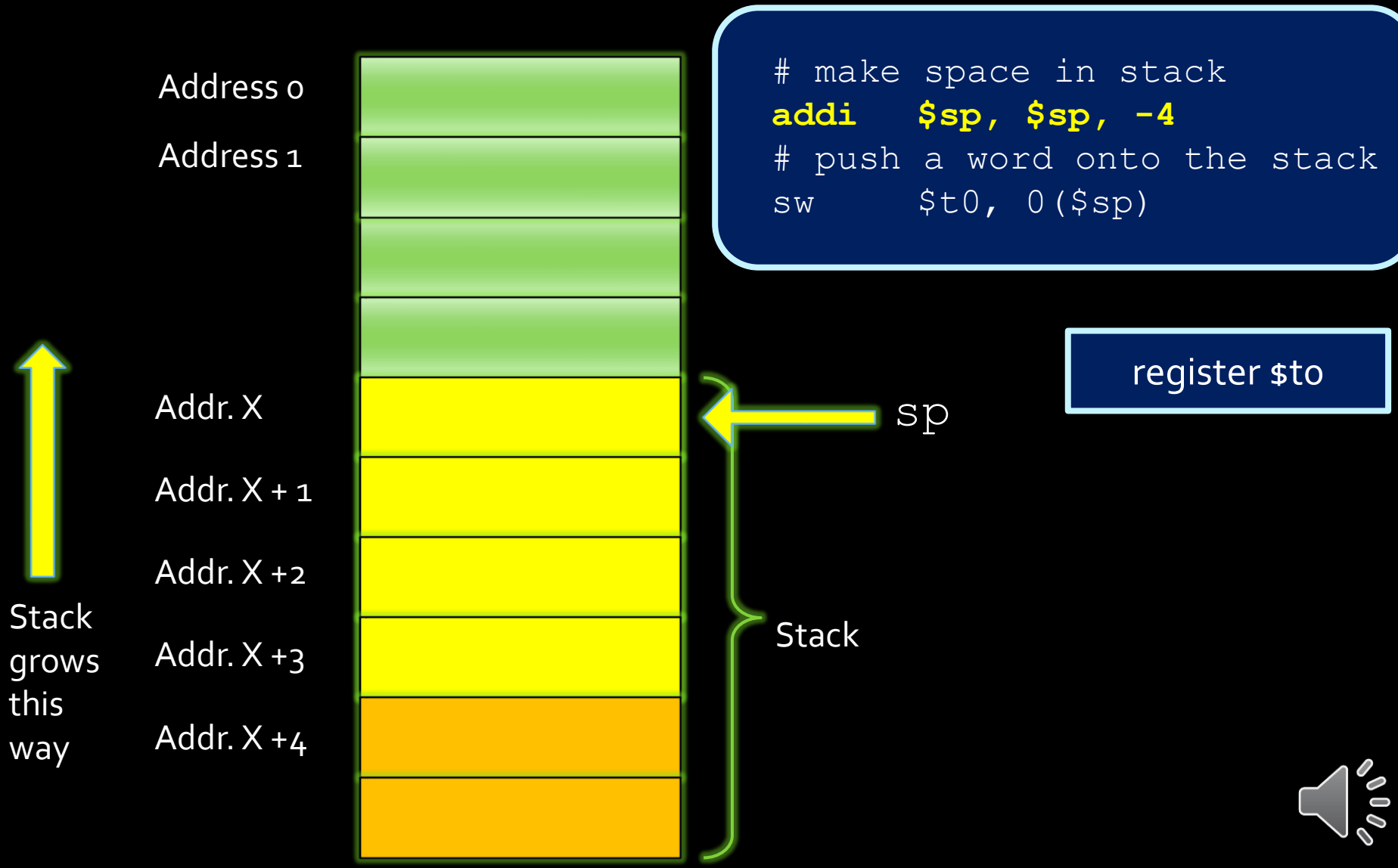
Pushing Values to the stack




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

register $to

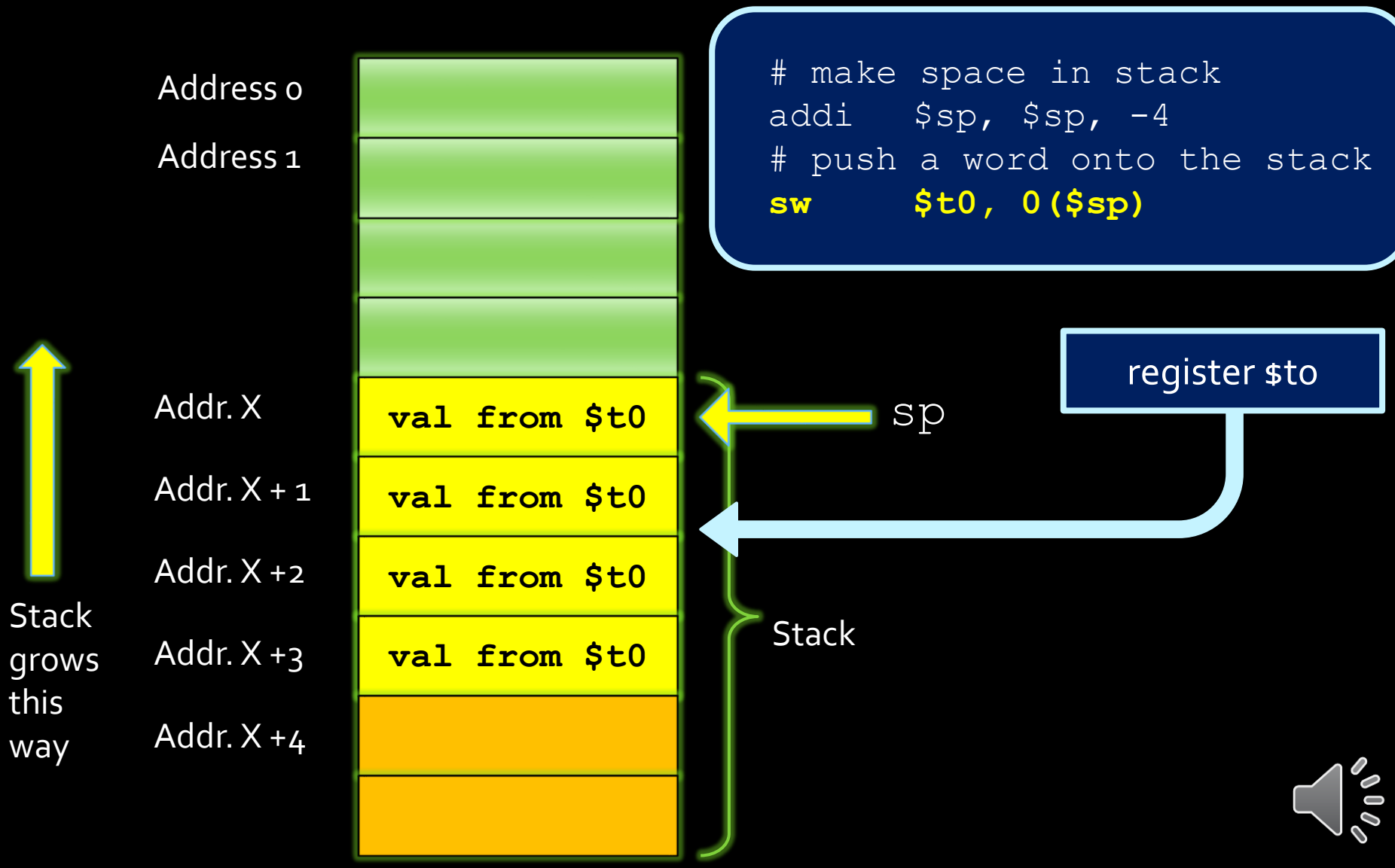
Pushing Values to the stack



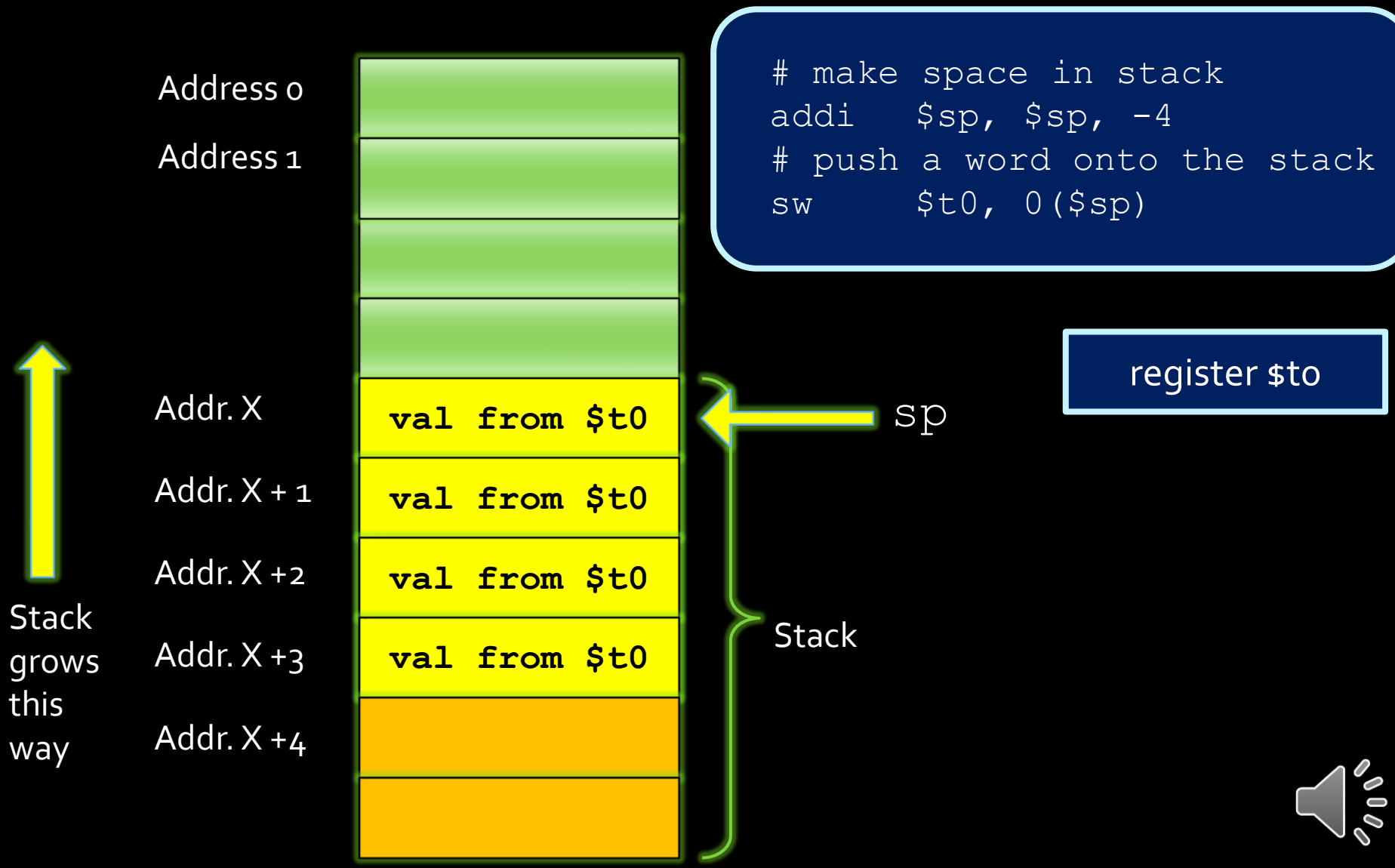

```
# make space in stack
addi   $sp, $sp, -4
# push a word onto the stack
sw     $t0, 0($sp)
```

register $to

Pushing Values to the stack



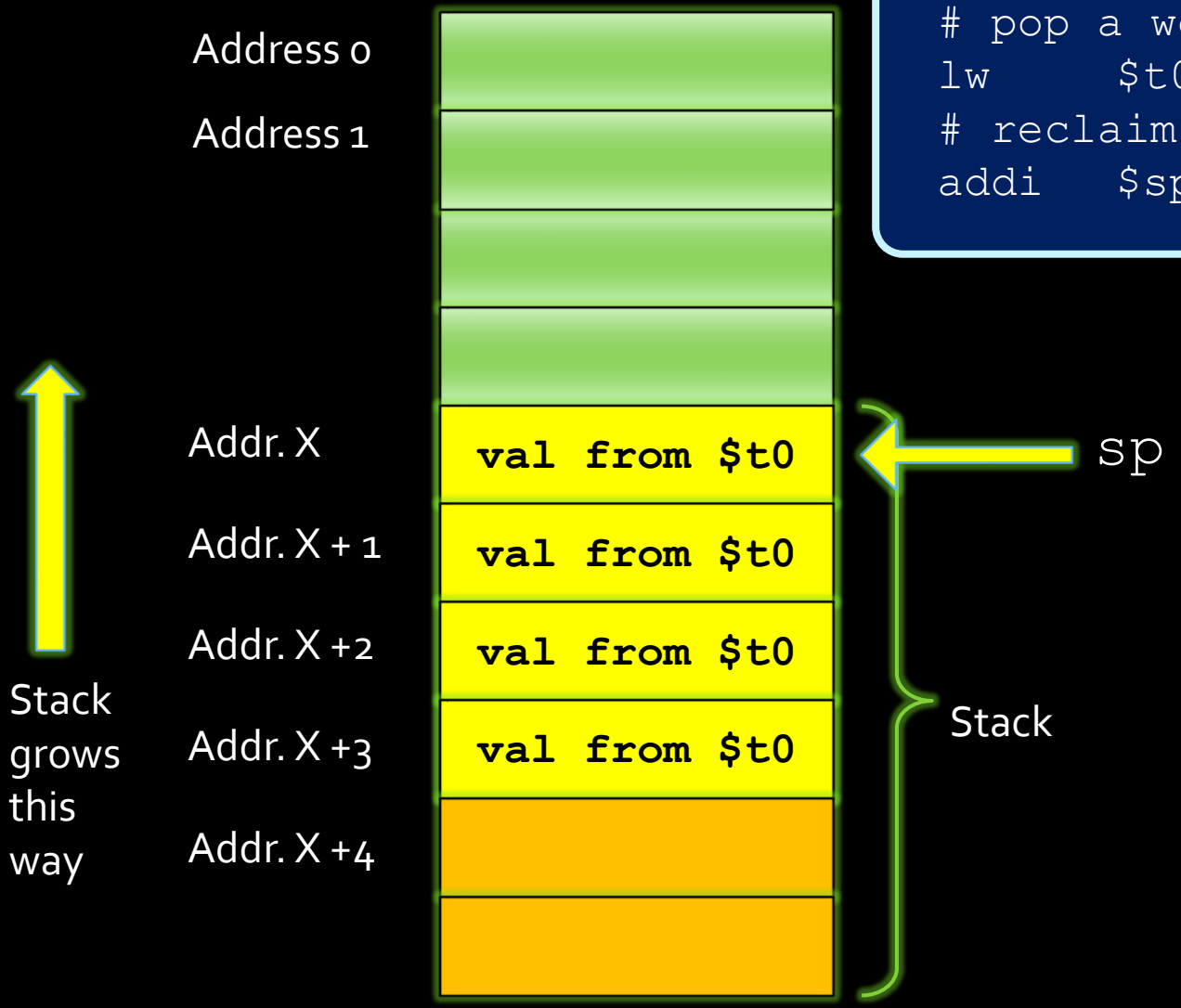
Pushing Values to the stack - After



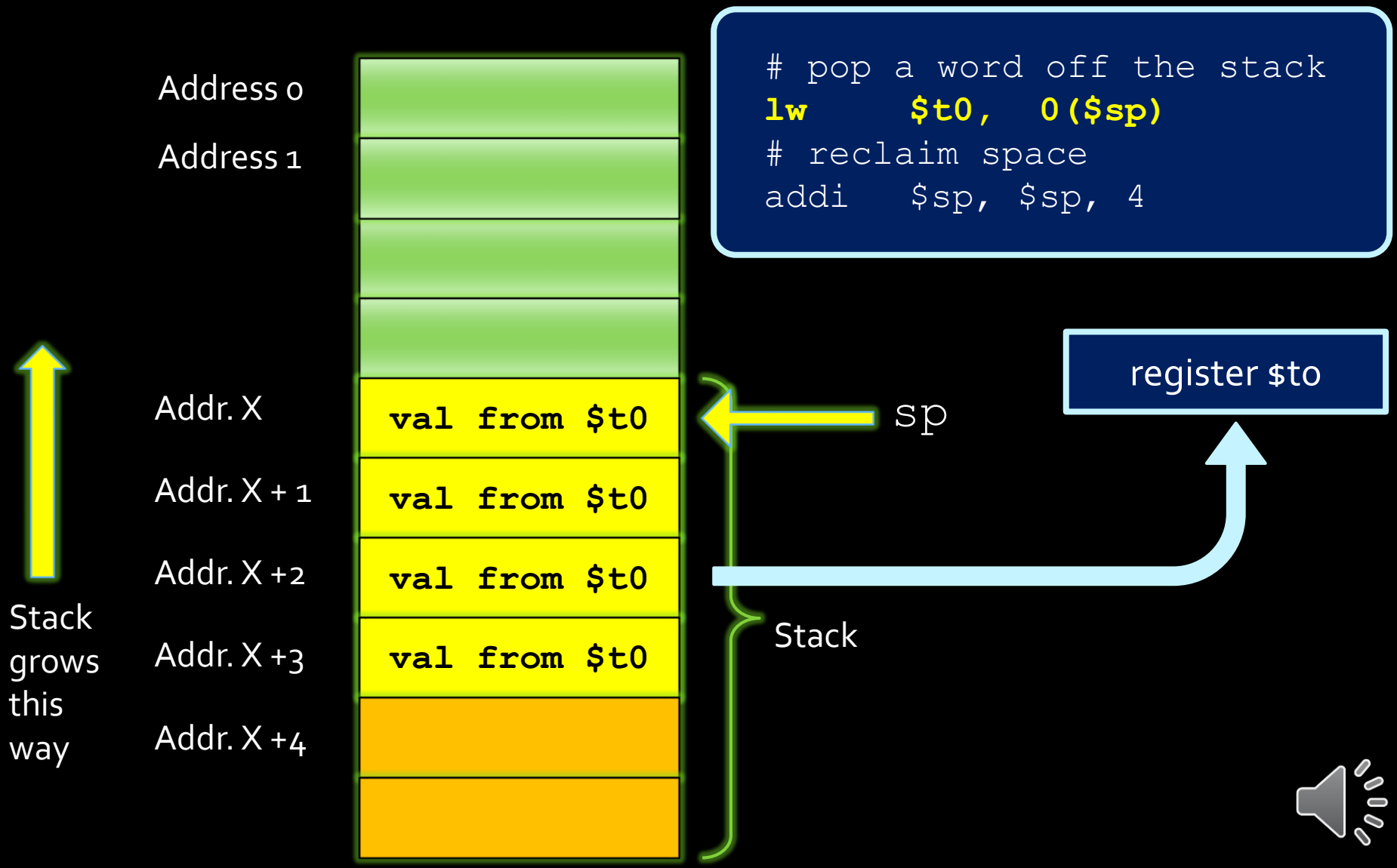
Popping Values off the stack

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

register $to



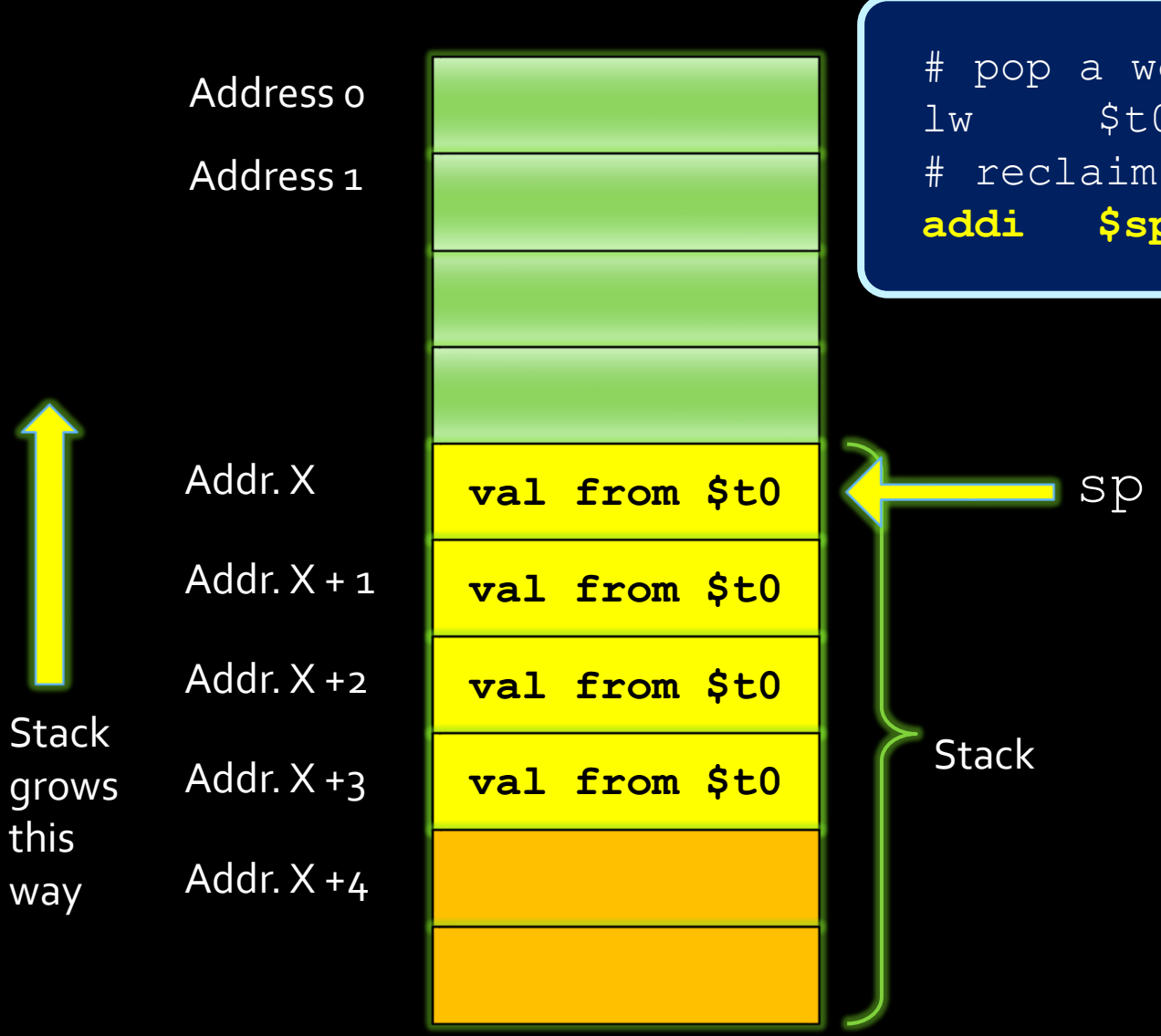
Popping Values off the stack



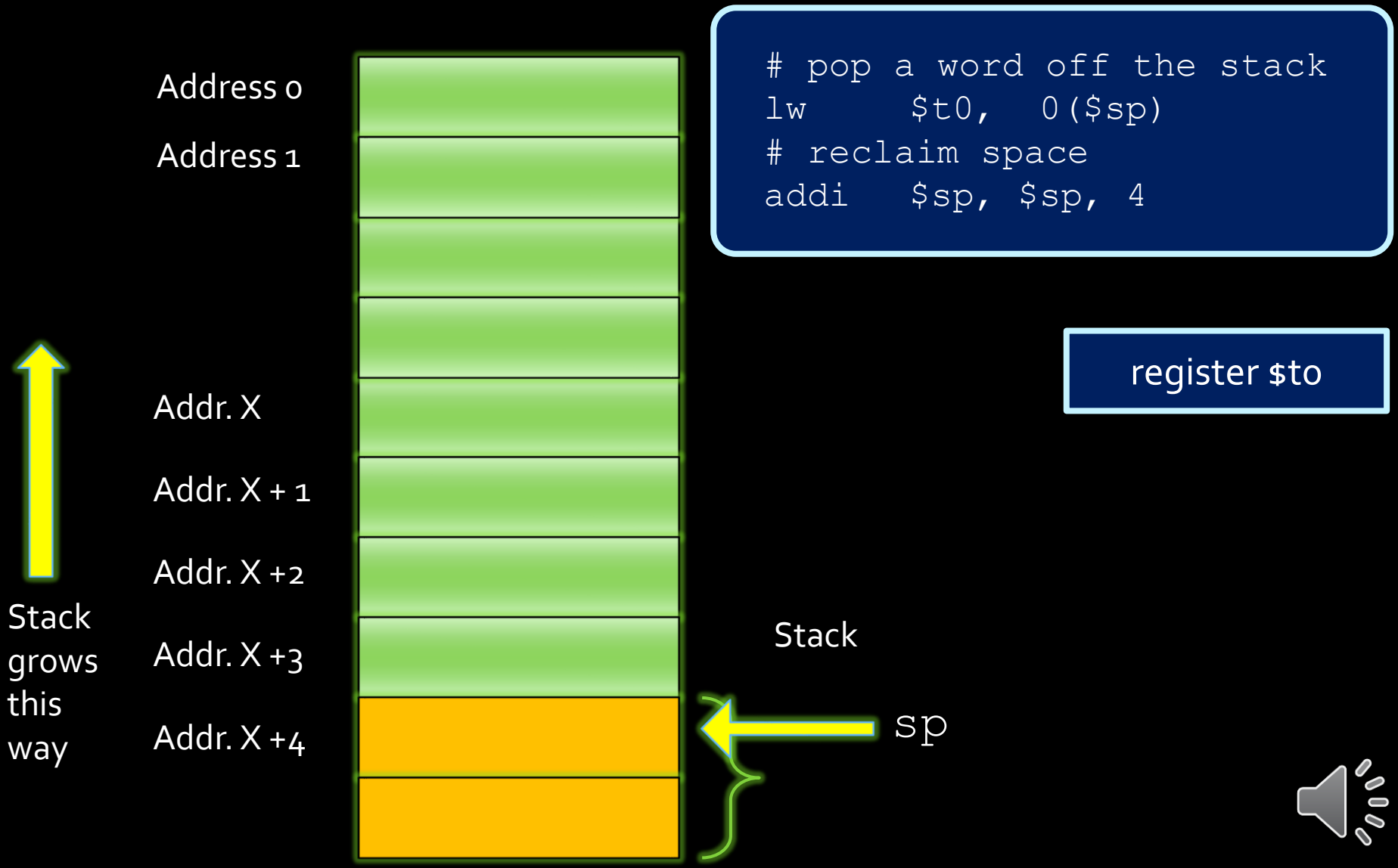
Popping Values off the stack

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

register $to



Popping Values off the stack - After



Simple Function

```
...
int r;
r = func(5,-2,-7);
...
```

```
void func(int a, int b, int c) {
    int res;
    res = a * b + c;
    return res;
}
```

- Let's convert this to assembly code!
- How do we call the function?
  - We'll have a label called `func`
  - Caller will jump to it.
  - How do we get back? That's a problem for later…

Simple Function

```
...
int r;
r = func(5,-2,-7);
...
```

```
void func(int a, int b, int c) {
    int res;
    res = a * b + c;
    return res;
}
```

- How do we pass parameters to the function?
  - Caller will push parameters on stack in order `a`, `b`, `c`.
  - We need to pop them from the stack in reverse order.
- Such design choices are called calling conventions:
  - Order of parameters, mechanism to pass values, etc.
  - Other choices are possible as long as we agree on it!

Simple Function

```
...
int r;
r = func(5,-2,-7);
...
```

```
void func(int a, int b, int c) {
    int res;
    res = a * b + c;
    return res;
}
```

- How do we pass return value to caller?
  - Put in on the stack!
- Another part of the calling convention.
  - A common alternative: use `$vo` and `$v1`.

Simple Function

```
...
int r;
r = func(5,-2,-7);
...
```

```
void func(int a, int b, int c) {
    int res;
    res = a * b + c;
    return res;
}
```

- How do we return to caller?
  - Where should we jump to?
  - Could just put PC onto stack
  - Better option: use special register $ra to store the return address before jumpying

How do we call a function?

```
...
int r;
r = func(5,-2,-7);
...
```

- `jal FUNC`
  - J-Type instruction.
  - Puts address of next instruction (PC+4) into register **`$ra`** (register $31, "return address")
  - Then jumps to label like a regular **`j`** instruction.
- Use `jal` after we've prepared the arguments (by pushing them on the stack)

How do we return from a function?

```
void func(int a, int b, int c) {
    int res;
    res = a * b + c;
    return res;
}
```

- `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)!

Function Calls – Cont'd

```
…
r = func(1,2,3);
next = 4;
…
```

(1) `jal func`
`$ra` set to PC of the next instruction (PC+4)

(4) Execution continues here

```
int func(int a, int b, int c) {
   int res;
   res = a * b + c;
   return res;
}
```

(2) Execution continues from here

(3) `jr $ra`

Putting it Together

- Caller calls Callee
  1. Caller pushes arguments onto the stack: A,B,C,…
  2. Caller stores PC into `$ra`, jumps to Callee
  3. Callee pops arguments from the stack (C, B, A…)
  4. Callee performs function
  5. Callee pushes return value onto stack
  6. Callee jumps to address stored in `$ra`
  7. Caller pops return value from stack
  8. Caller continues on its merry way

Caller (in main)

```
...
int r;
r = func(5,-2,-7);
...
```

```
main:  addi $t3, $zero, 5    # prepare A value
       addi $sp, $sp, -4     # push A onto the stack
       sw $t3, 0($sp)
       addi $t3, $zero, -2   # prepare B value
       addi $sp, $sp, -4     # push B onto the stack
       sw $t3, 0($sp)
       addi $t3, $zero, -7   # prepare C value
       addi $sp, $sp, -4     # push C onto the stack
       sw $t3, 0($sp)

       jal func              # call the function by
                             # putting PC+4 into $ra
                             # and jumping to function

       lw $t5, 0($sp)        # get result off the stack
       addi $sp, $sp, 4
```

Strategy

```
void func(int a, int b, int c) {
    int res;
    res = a * b + c;
    return res;
}
```

Initialization

- Pop parameter values from stack into registers in reverse order
  - Let us use `$t0,$t1,$t2` for `a, b, c`
- Also use `$t9` for temporary result.

Compute

- `$t9 = $to * $t1 + $t3`

At the end

- Push result from `$t9` on stack.
- Return to calling program – using `jr $ra`

Callee (Translated Function)

initialization

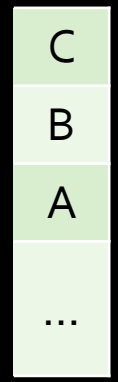
main algorithm

end

```
func: lw $t2, 0($sp)        # pop C off the stack (it's a
      addi $sp, $sp, 4      # (stack, so c will be first)
      lw $t1, 0($sp)        # pop B off the stack
      addi $sp, $sp, 4      #
      lw $t0, 0($sp)        # pop A off the stack
      addi $sp, $sp, 4      #


      mult $t0, $t1         # compute A*B
      mflo $t9
      add $t9,$t9,$t2       # add C

      addi $sp, $sp, -4     # push result on the stack
      sw $t9, 0($sp)        #

      jr $ra                # return to caller
```

Optimization

Save instructions by adding to $sp once at the end instead of after every pop.




```
func:  lw $t2, 0($sp)        # pop C off the stack
       lw $t1, 4($sp)        # pop B off the stack
       lw $t0, 8($sp)        # pop A off the stack
       addi $sp, $sp, 12     # reclaim space


       mult $t0, $t1         # compute A*B
       mflo $t9
       add $t9,$t9,$t2       # add C

       addi $sp, $sp, -4     # push result on the stack
       sw $t9, 0($sp)        #

       jr $ra                # return to caller
```

Reflections

- We've seen at least two ideas on how to implement function calls:
  - Use `$a0 - $a3` for arguments, `$v0` and `$v1` for return values.
  - Push on stack
- There are many other variants.
  - For example, should caller or callee pop variables?
  - Or using registers AND the stack.
- These are called calling conventions.

Reflections

- Functions calls are not free.
  - Must manipulate registers.
  - Read and write memory (stack)
  - Jump to another instructions
  - These have performance implications!
- Different calling convention trade off generality vs performance
  - Push to stack is general but slow
  - $a0-$a3 is faster but only supports 4 params and no function nesting (more next week)

Common Calling Conventions

- For us: always push all arguments and return values to the stack and pop them when needed.
  - We'll tell you if we want something else.
- Very common: combine registers and stack:
  - Use `$a0` to `$a3` registers for first four arguments (in that order).
    - First argument in `$a0`, second in `$a1`, and so on.
  - Any additional arguments are pushed on the stack.
  - Use `$v0`, `$v1` for return value.
    - Seldom need more (certainly not in C)

Food For thought

- How do we pass an array parameter?
  - Come to review.
- What happens if we call f and f then calls g?
  - Hint: think about $ra
- What about recursion?
- How do we handle local variables?

```
void func(int a[], int n)
{
...
}
```

```
void func(int a) {
    ...
    g()
    ...
}
```

```
void func(int a) {
    int x, arr[64];
    ...
}
```

It’s not Over

Next two week – more on functions:

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