Week 9 Review

W9 Material

- OS services (syscalls)
- Memory Instructions
  - Alignment and Endianess
- Structs and Arrays
- Stack
- Function calls

syscall example

```
.text
.globl main
main:
        # Read a number (result will be in $v0)
        li $v0, 5
        syscall

        addi $t0, $v0, 1     # $t0 = $v0 + 1

        # Print result
        li $v0, 1
        move $a0, $t0        # it will print the number in $a0
        syscall

        # End program
        li $v0, 10
        syscall
```

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”.

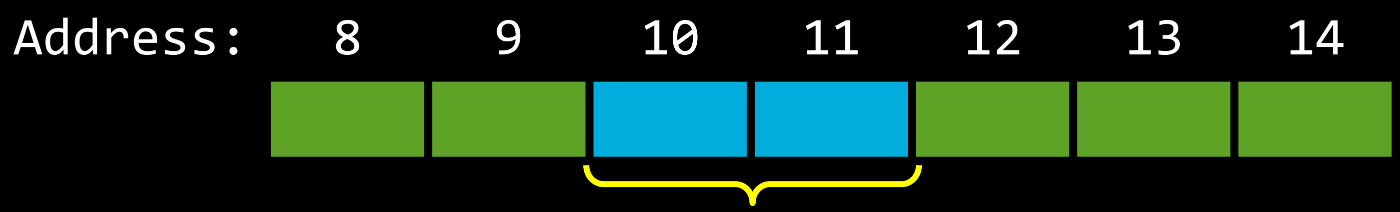
More Pseudo-instructions

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

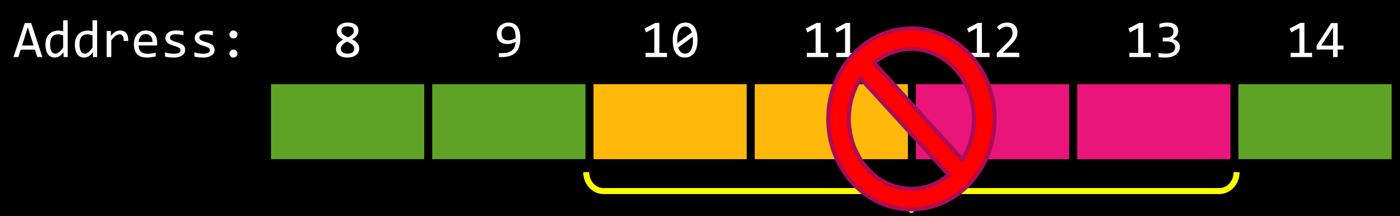
- Remember: these aren't really MIPS instructions
  - Simplifications of multiple instructions.
  - But they make life way easier.
- We already saw `li` (`lui` followed by `ori`)
- `la` loads the address of a label.
  - Implemented similarly to `li`

Addresses Must be Aligned

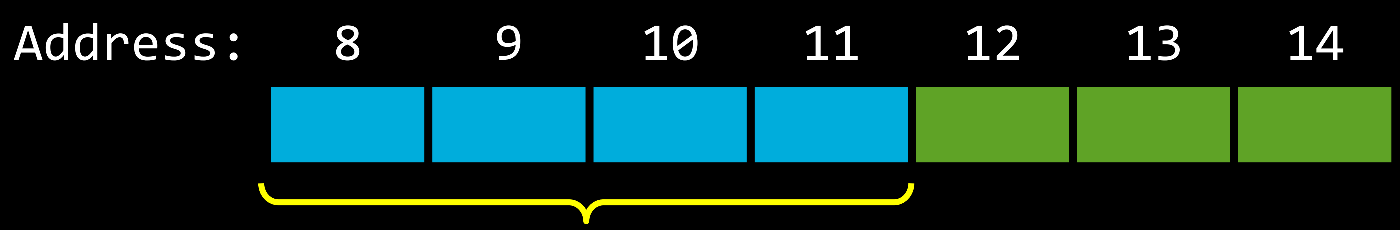
- Access to half-word at address 10 is aligned



- Access to word at address 10 is unaligned



- Access to word at address 8 is aligned



Storing 0x1234ABCD

- Big Endian ("big end first")
  - The most significant byte of the word is stored first (i.e., at address X). The 2nd most significant byte at address X+1 and so on.

Address	Byte
X	12
X + 1	34
X + 2	AB
X + 3	CD

- Little Endian ("little end first")
  - The least significant byte of the word is stored first (i.e., at address X). The 2nd least significant byte at address X+1 and so on.

Address	Byte
X	CD
X + 1	AB
X + 2	34
X + 3	12

Arrays in Assembly

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

- In assembly arrays are just a range of memory.
- Access arrays using address of the first element.
- To access element i in the array:
  - Start with the address of the first element
  - Add an offset (distance) in bytes from that address.
  - address of i = address of first element + i * (size of an element)
- Example: address of H[3] = address of `H[0] + 3*2`

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

Offset = 6 bytes = 3 elements

Example: A struct program

```
struct {
        int a;     s+0
        int b;     s+4
        int c;     s+8
} s;

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

address: s+0, s+4, s+8

```
.data
s:          .space   12
.text
.globl main
main:       la       $t0, 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)
```

- Like arrays, a bunch of bytes in memory.
- Unlike arrays, not all fields have the same type.
  - We have to guarantee correct alignment of all fields.

Padding

- Add padding: empty (unused) bytes between `a` and `c` so it is aligned for its type.
- Size of struct `s` is now 8 bytes.
  - We also make sure the struct initial address is word-aligned.
- A tradeoff of speed vs space.
  - Code is faster but uses more space.
  - Alternative: slower but use less memory.
    - Work with `y` using 2 half-word accesses or 4 byte accesses.
- Another option: put y before x.
  - Not always possible

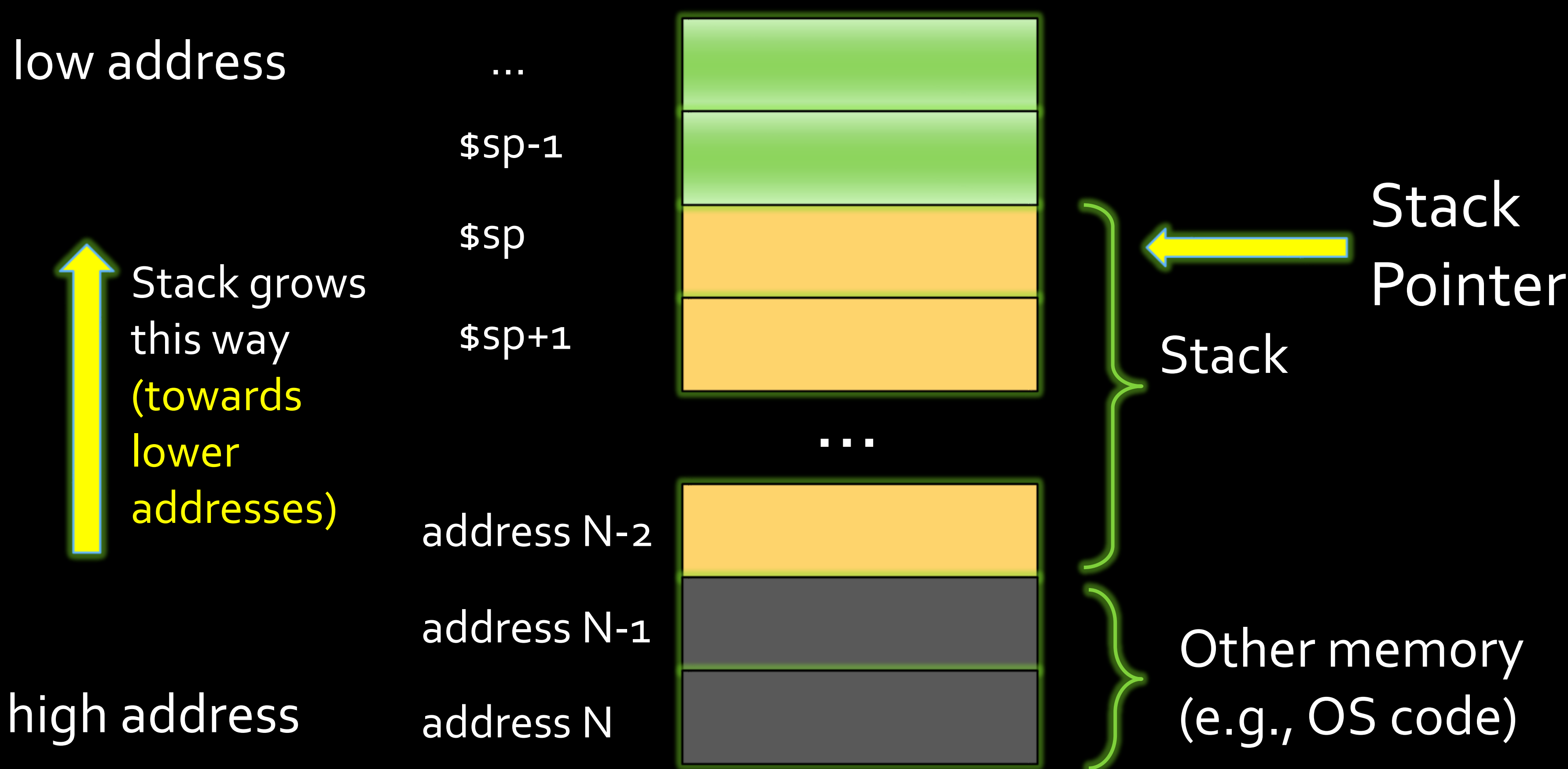
```
struct {
        char x;
        int y;
} s;

s.x = 5;
s.y = -7;
```

Address	Contents
0x1000 (s)	x
0x1001 (s+1)	padding
0x1002 (s+2)	padding
0x1003 (s+3)	padding
0x1004 (s+4)	y
0x1005 (s+5)	y
0x1006 (s+6)	y
0x1007 (s+7)	y

Function Call

- Call function with `jal LABEL`
- Return from function with `jr $ra`
- To pass parameters and return values:
  - In registers: use `$a0-$a3` and `$v0-$v1`
  - On stack: an area in memory where we can put things on top (push) and take off things (pop)
    - Our choice.

The Stack


low address
...
$sp-1
$sp
$sp+1
Stack grows this way (towards lower addresses)
...
address N-2
address N-1
address N
high address
Stack Pointer
Stack
Other memory (e.g., OS code)

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
```

Function Calls

- 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 the PC into $ra
                             # jump to function

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

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
```

Question 1a

- Write a piece of code to compute:
  `$t2 = max of $t0 and $t1`
  - Assume values a,b are already in `$t0, $t1`

```
# input values are in $t0, $t1, output will be in $t2
      ble $t0,$t1, else          # if a<=b we jump to else
      add $t2, $t0, $zero        # a>b so set $t2 to $t0
      j end
else: add $t2,$t1,$zero          # a<=b so set $t2 to $t1
end:
```

Question 1b

- Convert the previous code to function max(a,b)
  - Get a, b parameters from stack, result in return value

```
max:   lw $t1, 0($sp)               # first pop b from stack
       addi $sp, $sp, 4
       lw $t0, 0($sp)               # now pop a from stack
       addi $sp, $sp, 4

# input values are in $t0, $t1, output will be in $t2
       ble $t0,$t1, else            # if a<=b we jump to else
       add $t2, $t0, $zero          # a>b so set $t2 to $t0
       j end
else:  add $t2,$t1,$zero            # a<=b so set $t2 to $t1
end:   addi $sp, $sp, -4            # push result onto stack
       sw $t2, 0($sp)
       jr $ra                       # jump back to caller
```

Question 1c

- Call the function you just wrote from main!

Question 1c

- Call the function you just wrote from main!

```
main: addi, $t4, $zero, -3    # prepare first value
      addi $sp, $sp, -4       # make space on stack
      sw $t4, 0($sp)          # put on stack
      addi, $t4, $zero, 9     # push second value onto
      addi $sp, $sp, -4       # the stack
      sw $t4, 0($sp)
      jal max                 # "call" the max function
      lw $t4, 0($sp)          # pop the result from the
      addi $sp, $sp, 4        # the stack
      # result is now in $t4
```

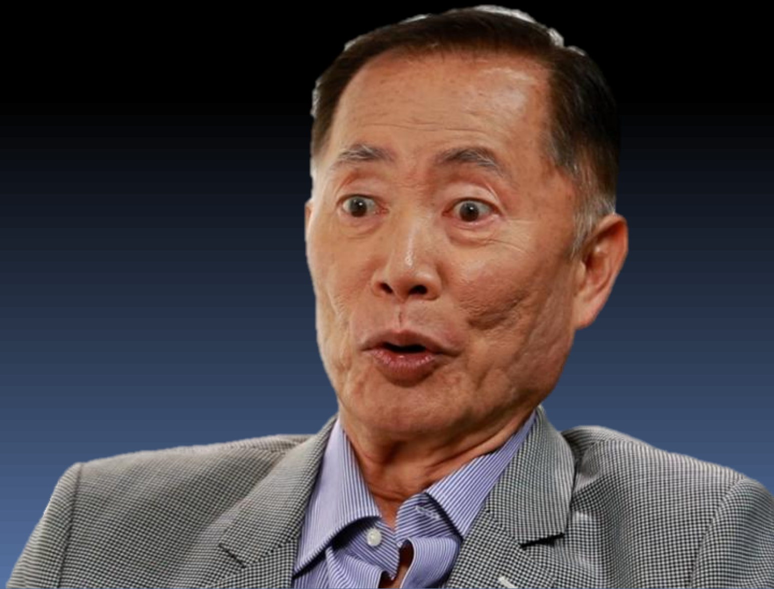
Project: Plan Ahead

- This project is challenging.
  - Harder than anything you've seen so far in CSCB58.
  - You will write hundreds of lines of assembly and comments.
  - Expect to spend at least 20-30 hours coding and debugging, probably more.
- Start **now**. You already know practically everything you need for the project.
- Plan your work and be organized
  - Handout has guidance and milestones to help you plan.
  - Following the milestones in order is a good idea.
  - You also have to do lab 5 in parallel!

More

- By Odin's beard, read the project handout carefully!
  - Yes, all of it.
  - Detailed and teaches you what you need to know.
  - Lots of tips and suggestions.
  - Information on features, marking, procedures.
- Start early
- Be creative, make the project yours!

Week 10, Nested functions and Stack Frames



Last Week

- Memory access:
  - Arrays
  - Structs
  - Alignment
  - Segments: .data , .text
- Functions:
  - Parameters
  - Stack
  - Return address
  - Calling conventions

```
.data
v1:     .word 52
a1:     .space 100

.text
la $t0, a1
lw $t2, 16($t0)
```

```
addi $sp, $sp, -4
sw $t2, 0($sp)

jal    SOME_FUNCTION

lw $t5, 0($sp)
addi $sp, $sp, 4
```

Warmup

- This functions has two parameters and two returned values

```
def sum_prod(a, b):
    s = a + b
    p = a * b
    return (s, p)
```

Calling sum_prod

- Given variables A, B, get sum_prod(A,B)
- Steps:
  - Declare variables
  - Load into registers
  - Push onto stack
  - Call function
  - Pop results from stack into registers

Calling sum_prod: declare vars

```
.data
A:      .word  7
B:      .word  5
```

Calling sum_prod: load values

```
.data
A:     .word  7
B:     .word  5

.text
main:  la $t0, A           # $t0 = address of A
       lw $t1, 0($t0)      # $t1 = value of A
       la $t2, B           # $t2 = address of B
       lw $t3, 0($t2)      # $t2 = value of B
```

Calling sum_prod: push and call

```
.data
A:      .word  7
B:      .word  5

.text
main:  la $t0, A            # $t0 = address of A
       lw $t1, 0($t0)       # $t1 = value of A
       la $t2, B            # $t2 = address of B
       lw $t3, 0($t2)       # $t2 = value of B

       addi $sp, $sp, -4    # push A onto the stack
       sw $t1, 0($sp)
       addi $sp, $sp, -4    # push B onto the stack
       sw $t3, 0($sp)
       jal sumprod          # “call" the sign function
```

Calling sum_prod: pop results

```
.data
A:      .word  7
B:      .word  5

.text
main:  la $t0, A            # $t0 = address of A
       lw $t1, 0($t0)       # $t1 = value of A
       la $t2, B            # $t2 = address of B
       lw $t3, 0($t2)       # $t3 = value of B

       addi $sp, $sp, -4    # push A onto the stack
       sw $t1, 0($sp)
       addi $sp, $sp, -4    # push B onto the stack
       sw $t3, 0($sp)
       jal sum_prod         # "call" the sign function
       lw $t5, 0($sp)       # pop the sum A+B off the stack
       addi $sp, $sp, 4
       lw $t6, 0($sp)       # pop the product A*B off stack
       addi $sp, $sp, 4
```

The Stack is FILO / LIFO

- First in, last out.
- Equivalently:
  Last in, first out (LIFO).
- If you push A, B, C (in this order)…
- …when you pop you get C, B, A

sum_prod: implementation

- To implement sum_prod
  - (decide on registers)
  - Pop arguments off stack
  - Do the computation
  - Push return values
  - Return to caller

Implement sum_prod: arguments

```
# sum_prod: (A, B) -> (A+B, A*B)
# $t0=A, $t1=B, $t3=A+B, $t4=A*B

sum_prod:     lw $t1, 0($sp)        # pop B off the top of
              addi $sp, $sp, 4      # the stack first
              lw $t0, 0($sp)        # now we pop A
              addi $sp, $sp, 4
```

Implement sum_prod: compute

```
# sum_prod: (A, B) -> (A+B, A*B)
# $t0=A, $t1=B, $t3=A+B, $t4=A*B

sum_prod:     lw $t1, 0($sp)        # pop B off the top of
              addi $sp, $sp, 4      # the stack first
              lw $t0, 0($sp)        # now we pop A
              addi $sp, $sp, 4

              add $t3, $t0, $t1     # $t3 = A+B
              mult $t0, $t1         # compute A*B
              mflo $t4              # store the result in $t4
#(note we are assuming 32 bit result here!)
```

Implement sum_prod: return

```
# sum_prod: (A, B) -> (A+B, A*B)
# $t0=A, $t1=B, $t3=A+B, $t4=A*B

sum_prod:     lw $t1, 0($sp)        # pop B off the top of
              addi $sp, $sp, 4      # the stack first
              lw $t0, 0($sp)        # now we pop A
              addi $sp, $sp, 4

              add $t3, $t0, $t1     # $t3 = A+B
              mult $t0, $t1         # compute A*B
              mflo $t4              # store the result in $t4
#(note we are assuming 32 bit result here!)

end:          addi $sp, $sp, -4     # first push A*B on stack
              sw $t4, 0($sp)        # so it comes out second
              addi $sp, $sp, -4     # now push A+B onto stack
              sw $t3, 0($sp)        # so it comes out first

              jr $ra                # jump back to caller
```

What About Nested Calls?

- How do we call a function from inside another function?

```
int f(int a, int b)
{
    return a + b;
}

int g(int x)
{
        return 2 * f(x, 5);
}
```

- Problem 1: `jal` will overwrite `$ra`
- Problem 2: we need values in registers

→ Answers in next part!

Functions Calling Functions

Problems calling `f` from inside `g`:

1. `main` calls `g`
2. `g` calls `f`
   - Overwrites return address in `$ra`.
3. `f` executes
   - Might overwrite registers needed by `g`.
4. `f` returns to `g`
5. `g` returns to main
   - But `$ra` was overwritten in step 2…

Calling From Inside Function

- Assume we already have max(a,b)
- We want to implement `max3(a,b,c)`
- Easy, just call `max` twice:
  - `tmp = max(a, b)`
  - `res = max(tmp, c)`
  - `return res`
- max pseudo code:
  - pop a, b into \$t0, \$t1
  - If \$t0 > \$t1 set \$t2 = \$t0 else \$t2 = \$t1
  - Push \$t2 onto stack

max(a,b)

```
max:  lw $t1, 0($sp)              # first pop b from stack
      addi $sp, $sp, 4
      lw $t0, 0($sp)              # now pop a from stack
      addi $sp, $sp, 4

# input values are in $t0, $t1, output will be in $t2
      ble $t0,$t1, else           # if a<=b we jump to else
      add $t2, $t0, $zero         # a>b so set $t2 to $t0
      j end
else: add $t2,$t1,$zero           # a<=b so set $t2 to $t1
end:  addi $sp, $sp, -4           # push result onto stack
      sw $t2, 0($sp)
      jr $ra                      # jump back to caller
```

`max3(a,b,c)` in "Assembly"

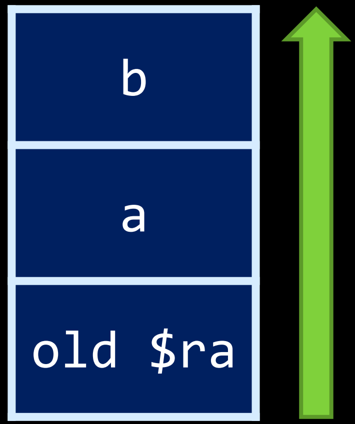
- Pop a, b, c into registers `$t0`, `$t1`, `$t2`
- Push a, b onto stack
- Call max (`jal max`)
- Pop partial max into `$t3`
- Push `$t2`, `$t3` onto stack
- Call max again
- Pop final max into `$t4`
- Push `$t4` final max
- Return to caller (jr `$ra`)

Problem 1:
max overwrites
`$t2` internally

Problem 2:
`$ra` was
overwritten by
`jal max`

Saving `$ra`

- When calling function `f(a,b)` from inside function g we execute `jal f`
- This overwrites return address `$ra`
- We need to preserve it, but where?
- Stack to the rescue:
  - Push old value of `$ra` onto the stack.
  - Push arguments for f
  - Call f
  - Pop return value
  - Pop old value of `$ra`



max3(a,b,c) in "Assembly"

- Pop a, b, c into registers $t0, $t1, $t2
- Push $ra
- Push a, b onto stack
- Call max (`jal max`)
- Pop partial max into $t3
- Pop $ra
- Push $ra
- Push $t2, $t3 onto stack
- Call max again
- Pop final max into $t4
- Pop $ra
- Push $t4 final max
- Return to caller (jr $ra)

This is a little silly.

`max3(a,b,c)` in “Assembly”

- Pop a, b, c into registers `$t0`, `$t1`, `$t2`
- Push `$ra`
- Push a, b onto stack
- Call max (`jal max`)
- Pop partial max into `$t3`
- Push `$t2`, `$t3` onto stack
- Call max again
- Pop final max into `$t4`
- Pop `$ra`
- Push `$t4` final max
- Return to caller (`jr $ra`)

If we need to call another function: push `$ra` to the stack at the beginning of the function, and pop at the end

Wait a Minute…

- We also need to preserve $t2 since it holds c

```
# part of the code for max
    ...
    ble $t0,$t1, else
    add $t2, $t0, $zero
    ...
```

- Solution:
  Push $t2 on stack along with $ra and pop it after the max function returns

max3(a,b,c) in “Assembly”

- Pop a, b, c into registers $t0, $t1, $t2
- Push $ra
- Push $t2 (we’ll remember to pop $t2 before $ra!)
- Push a, b onto stack
- Call max (`jal max`)
- Pop partial max into $t3
- Pop $t2
- Push $t2, $t3 onto stack
- Call max again
- Pop final max into $t4
- Pop $ra
- Push $t4 final max
- Return to caller (jr $ra)

Preserving Register Values

- We've already demonstrated why we'd need to push `$ra` and `$t2` onto the stack when calling function from another function.
- What about the other registers?
- How do we know that a function we called didn't overwrite registers that we were using?
  - Remember there is only one register file!

Specify caller vs. callee in calling conventions.

Calling Conventions

- We've seen at least two calling conventions that specify how to implement function calls:
  - Use `$a0` - `$a3` , `$v0` and `$v1`, and so on.
  - Push on stack (ours)
- There are many other variants.
  - For example, should caller or callee pop variables?
- We will now extend our calling convention to specify who can use which registers.

Calling Conventions

A function can be both a caller and a callee (e.g. recursion).

- Caller vs. Callee
  - Caller is the function calling another function.
  - Callee is the function being called.
- We separate registers into:
  - Caller-Saved registers (`$t0-$t9`)
    - Also called "unsaved (or temporary) registers".
  - Callee-Saved registers (`$s0-$s7`)
    - Also called "saved registers"

Register Saving Conventions

Push to stack just before you call another function. Restore them immediately after.

- Caller-Saved registers
  - Registers `$t0-$t9`: temporaries
  - Also registers `$a0-$a3` and `$v0-$v1`
  - Registers that the caller should save to the stack before calling a function (if the caller will need their value).
  - Functions are free to clobber (overwrite) the registers.

Register Saving Conventions

Push to stack just before you call another function. Restore them immediately after.

- Caller-Saved registers
  - Registers `$t0-$t9`: temporaries
  - Also registers `$a0-$a3` and `$v0-$v1`
  - Registers that the caller should save to the stack before calling a function (if the caller will need their value).
  - Functions are free to clobber (overwrite) the registers.
- Callee-Saved registers
  - Registers `$s0-$s7` (saved temporaries) and `$ra`
  - The callee must save these registers and later restore them, if it's going to modify them.
  - Push them to the stack near the beginning of your function body and restore them just before you return!

Caller-Saved ($t0-$t9) vs. Callee-Saved ($s0-$s7) Registers

Caller code

- Using $to-$t9 and you care for their values?
  - Push them to the stack just before you make a function call and restore them immediately after the calling site.
  - If you don't care about the value, no need to do anything.
  - (also $ao-$a3, $vo-$v1)
- Using $so-$s7 or $ra?
  - No action needed. It is the responsibility of the callee to ensure these registers are not modified.

Callee code

- Using $to-$t9?
  - (also $ao-$a3, $vo-$v1)
  - No action needed. It it the responsibility of the caller to ensure there registers are not modified.
- Using $so-s7 or $ra (jal)?
  - You need to ensure these registers are not modified.
  - If you plan to modify them, push them to the stack in the beginning of your function and restore them in the very end just before the jr $ra.

If a function is both a caller and a callee, it will fall under both categories.

We Can Do Nested Calls

- For nested calls:
  - Save $ra
  - Save needed registers
  - jal
  - Restore registers
  - Restore $ra
- We now have the power of recursion!
  - In part C