Review

- When a register is filled with a *signed* value of fewer bits – how are the more significant bits treated on MIPS?
- What is `$at`?
- Explain the difference between a computer's assembly language and its machine language
- When a program has a test

  ```
  if (x < 5) …
  ```

  the test is usually coded as `bge` … why?
- Every MIPS instruction is ____ bits long
- What does "opcode" mean?

R-type format

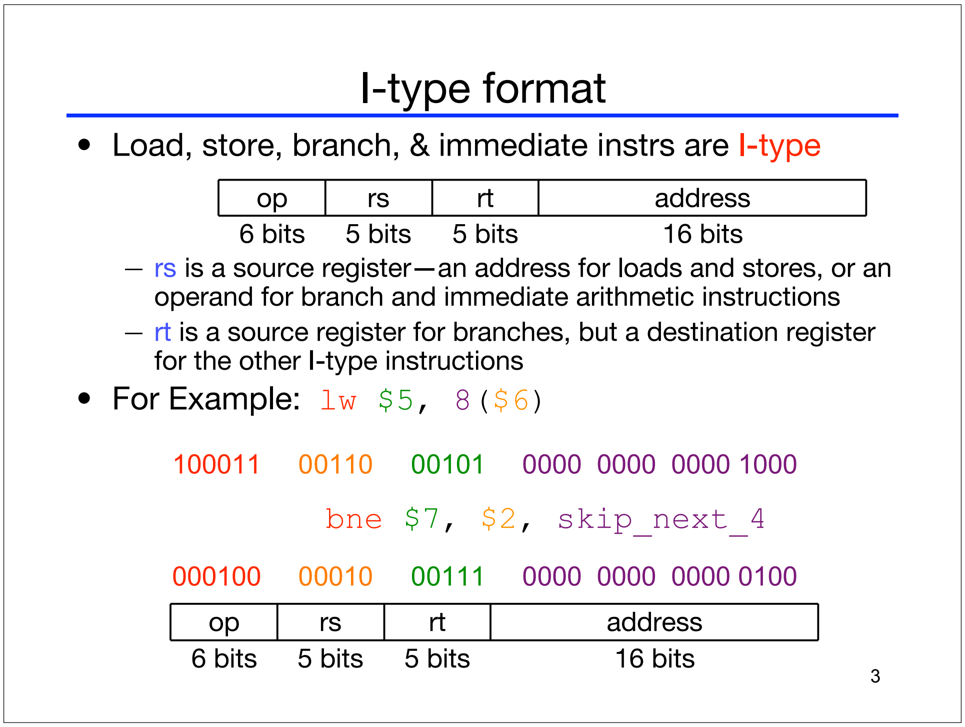
- Recall

op	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

  - op is an operation code or opcode that selects a specific operation
  - rs and rt are the first and second source registers
  - rd is the destination register
  - shamt is "shift amount" and is only used for shift instructions
  - func is used together with op to select an arithmetic instruction
- For example: add $4, $3, $2

000000 00011 00010 00100 00000 10 0000

op	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits



I-type format

- Load, store, branch, & immediate instrs are I-type

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits

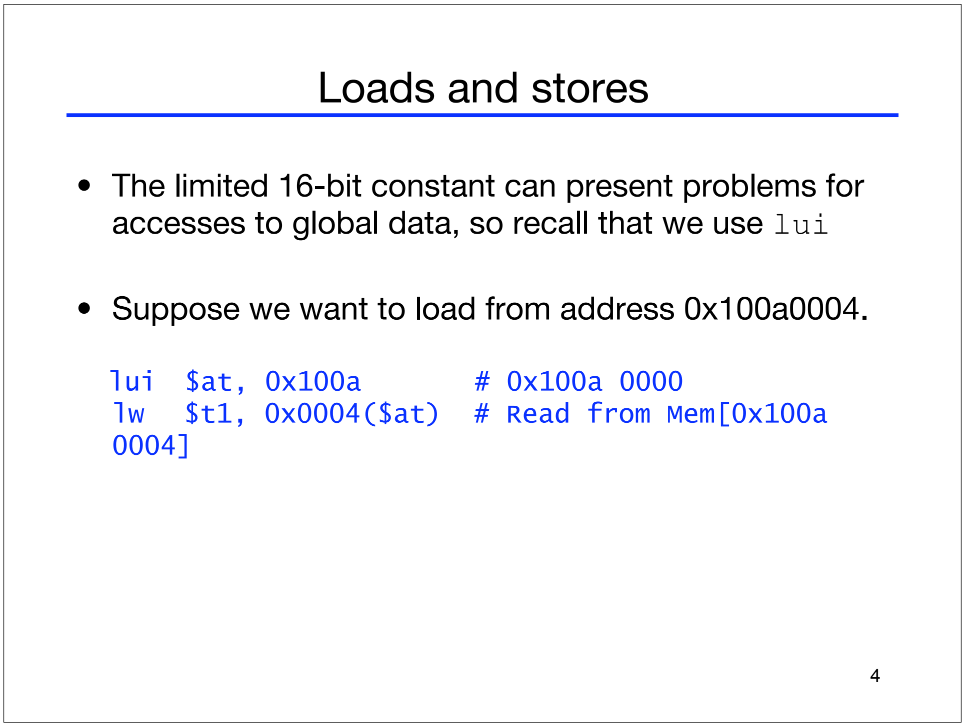
  - rs is a source register—an address for loads and stores, or an operand for branch and immediate arithmetic instructions
  - rt is a source register for branches, but a destination register for the other I-type instructions
- For Example: `lw $5, 8($6)`

100011 00110 00101 0000 0000 0000 1000

`bne $7, $2, skip_next_4`

000100 00010 00111 0000 0000 0000 0100

op	rs	rt	address
6 bits	5 bits	5 bits	16 bits



Loads and stores

- The limited 16-bit constant can present problems for accesses to global data, so recall that we use `lui`

- Suppose we want to load from address 0x100a0004.

```
lui  $at, 0x100a        # 0x100a 0000
lw   $t1, 0x0004($at)   # Read from Mem[0x100a
0004]
```

Branches

- For branch instructions, the constant field is not an address, but an *offset* from the current PC or program counter (== next instruction address) to the target address: *relative addressing*

```
    beq            $at,  $0, L
     add           $v1, $v0, $0
     add           $v1, $v1, $v1
     j             Somewhere
L:                 add   $v1, $v0, $v0
```

- Since the branch target L is three *instructions* past the beq, the address field would contain 3. The whole beq instruction would be stored as:

000100	00001	00000	0000 0000 0000 0011
op	rs	rt	address

Larger Branch Constants

- Empirical studies of real programs show that most branches go to targets less than 32,767 instructions away—branches are mostly used in loops and conditionals, and programmers are taught to make code bodies short
- If you do need to branch further, you can use a jump with a branch. For example, if “far” is very far away, then the effect of:

```
        beq $s0, $s1, far
```

- Can be simulated with the following code

```
        bne $s0, $s1, near
        j   far
near:   …
```

- Again, the MIPS designers have taken care of the common case first

J-type format

- Finally, the jump instruction uses the J-type instruction format.

op	address
6 bits	26 bits

- The jump instruction has a *word* address, not an offset
  - ❖ Remember that each MIPS instruction is one word long, and word addresses must be divisible by four.
  - ❖ So instead of saying “jump to address 4000,” it’s enough to just say “jump to instruction 1000.”
  - ❖ A 26-bit address field allows for jumps to any address from 0 to 2^{28}-1
    - your MP solutions had better be smaller than 256MB
- For even longer jumps, the jump register, or jr, instruction can be used.

```
jr$ra   # Jump to 32-bit address in register $ra
```

Summary of Machine Language

- Machine language is the binary representation of instructions:
  - ❖ The format in which the machine actually executes them
- MIPS machine language is designed to simplify processor implementation
  - ❖ Fixed length instructions
  - ❖ 3 instruction encodings: R-type, I-type, and J-type
  - ❖ Common operations fit in 1 instruction
    - Uncommon (e.g., long immediates) require more than one

R	opcode	rs	rt	rd	shamt	funct
I	opcode	rs	rt	immediate		
J	opcode	target address				

Functions in MIPS

- We'll talk about the 3 steps in handling function calls:
  1. The program's flow of control must be changed.
  2. Arguments and return values are passed back and forth.
  3. Local variables can be allocated and destroyed.
- And how they are handled in MIPS:
  - ❖ New instructions for calling functions.
  - ❖ Conventions for sharing registers between functions.
  - ❖ Use of a stack.

Control flow in C

- Invoking a function changes the control flow of a program twice.
  1. Calling the function
  2. Returning from the function
- In this example the main function calls `fact` twice, and fact returns twice—but to *different* locations in main.
- Each time fact is called, the CPU has to remember the appropriate return address.
- Notice that main itself is also a function! It is called by the operating system when you run the program.

```
int main()
{
   ...
   t1 = fact(8);
   t2 = fact(3);
   t3 = t1 + t2;
   ...
}

int fact(int n)
{
   int i, f = 1;
   for (i = n; i >
   1; i--)
      f = f * i;
   return f;
}
```

Control flow in MIPS

- MIPS uses the jump-and-link instruction jal to call functions.
  - ❖ The jal saves the return address (the address of the *next* instruction) in the dedicated register $ra, before jumping to the function.
  - ❖ jal is the only MIPS instruction that can access the value of the program counter, so it can store the return address PC+4 in $ra.

```
jal fact
```

- To transfer control back to the caller, the function just has to jump to the address that was stored in $ra.

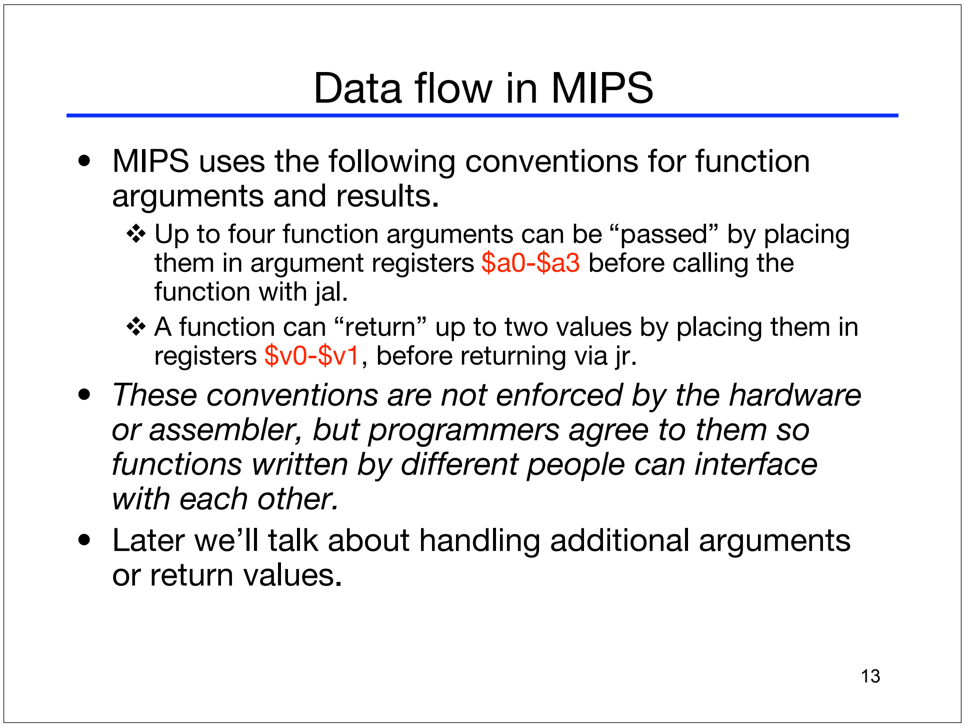
```
jr $ra
```

Data flow in C

- Functions accept arguments and produce return values.
- The blue parts of the program show the actual and formal arguments of the fact function.
- The purple parts of the code deal with returning and using a result.

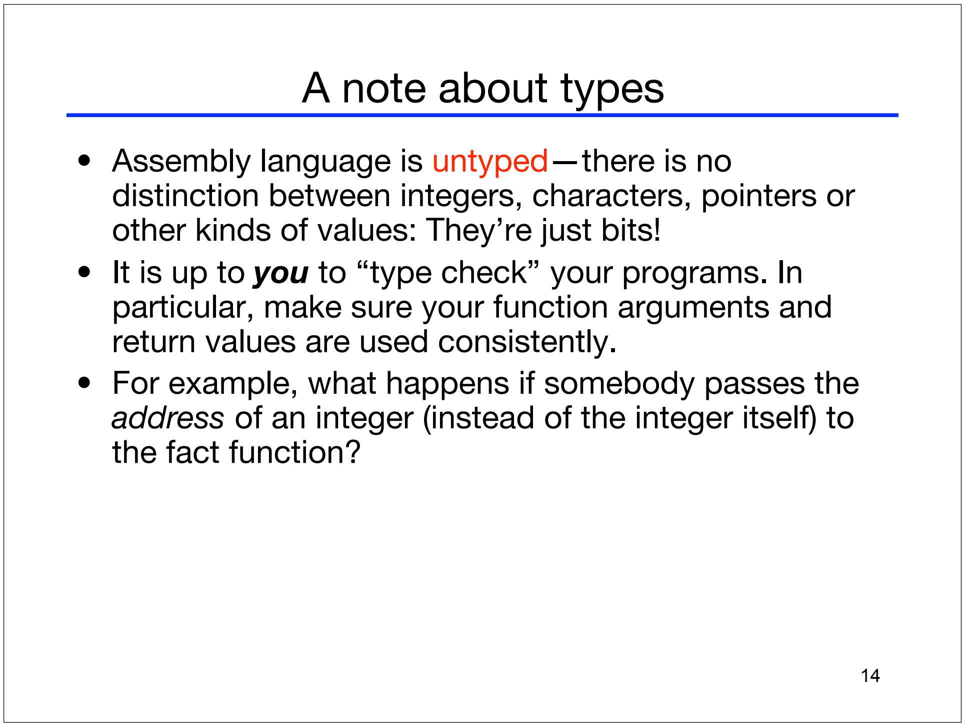
```
int main()
{
   ...
   t1 = fact(8);
   t2 = fact(3);
   t3 = t1 + t2;
   ...
}

int fact(int n)
{
   int i, f = 1;
   for (i = n; i > 1; i--)
      f = f * i;
   return f;
}
```



Data flow in MIPS

- MIPS uses the following conventions for function arguments and results.
  - ❖ Up to four function arguments can be “passed” by placing them in argument registers $a0-$a3 before calling the function with jal.
  - ❖ A function can “return” up to two values by placing them in registers $v0-$v1, before returning via jr.
- *These conventions are not enforced by the hardware or assembler, but programmers agree to them so functions written by different people can interface with each other.*
- Later we’ll talk about handling additional arguments or return values.



A note about types

- Assembly language is untyped—there is no distinction between integers, characters, pointers or other kinds of values: They’re just bits!
- It is up to ***you*** to “type check” your programs. In particular, make sure your function arguments and return values are used consistently.
- For example, what happens if somebody passes the *address* of an integer (instead of the integer itself) to the fact function?

The big problem so far

- There is a big problem here!
  - ❖ The main code uses $t1 to store the result of fact(8).
  - ❖ But $t1 is also used within the fact function!
- The subsequent call to fact(3) will overwrite the value of fact(8) that was stored in $t1.

Nested functions

- A similar situation happens when you call a function that then calls another function.
- Let's say A calls B, which calls C.
  - ❖ The arguments for the call to C would be placed in $a0-$a3, thus *overwriting* the original arguments for B.
  - ❖ Similarly, jal C overwrites the return address that was saved in $ra by the earlier jal B.

```
A:   ...
     # Put B's args in $a0-
     $a3
     jal B      # $ra = A2
A2:  ...
```

```
B:   ...
     # Put C's args in $a0-
     $a3,
     # erasing B's args!
     jal C      # $ra = B2
B2:  ...
     jr  $ra    # where does
                # this go???
```

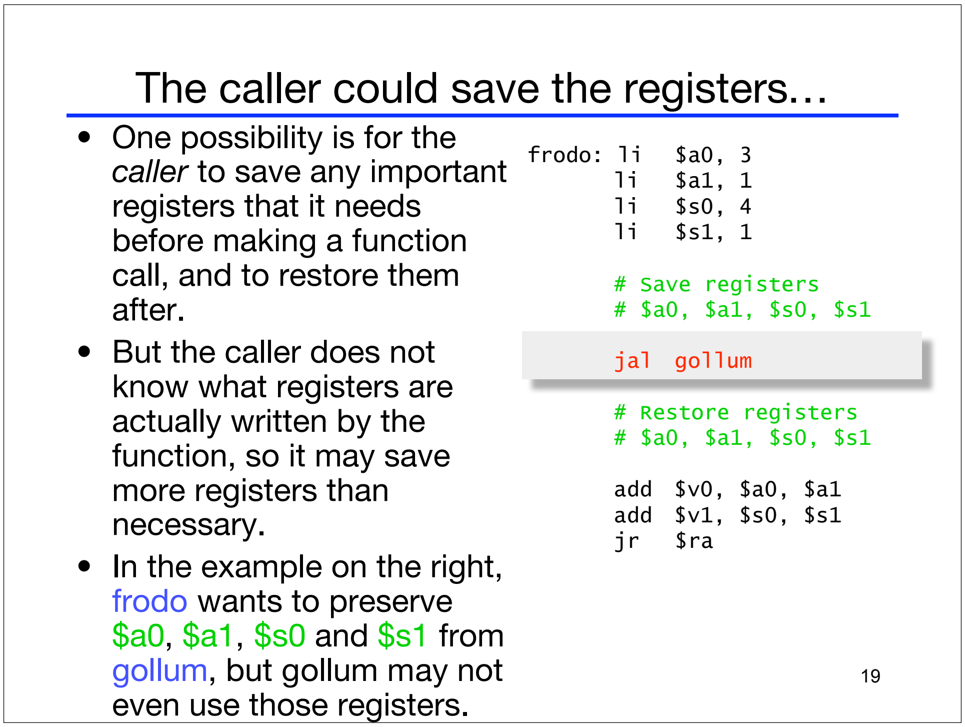
```
C:   ...
     jr  $ra
```

Spilling registers

- The CPU has a limited number of registers for use by all functions, and it’s possible that several functions will need the same registers.
- We can keep important registers from being overwritten by a function call, by saving them before the function executes, and restoring them after the function completes.
- But there are two important questions.
    - ❖ Who is responsible for saving registers—the caller or the callee?
    - ❖ Where exactly are the register contents saved?

Who saves the registers?

- Who is responsible for saving important registers across function calls?
    - ❖ The caller knows which registers are important to it and should be saved.
    - ❖ The callee knows exactly which registers it will use and potentially overwrite.
- However, in the typical “black box” programming approach, the caller and callee do not know anything about each other’s implementation.
    - ❖ Different functions may be written by different people or companies.
    - ❖ A function should be able to interface with any client, and different implementations of the same function should be substitutable.
- So how can two functions cooperate and share registers when they don’t know anything about each other?



The caller could save the registers…

- One possibility is for the *caller* to save any important registers that it needs before making a function call, and to restore them after.
- But the caller does not know what registers are actually written by the function, so it may save more registers than necessary.
- In the example on the right, frodo wants to preserve $a0, $a1, $s0 and $s1 from gollum, but gollum may not even use those registers.

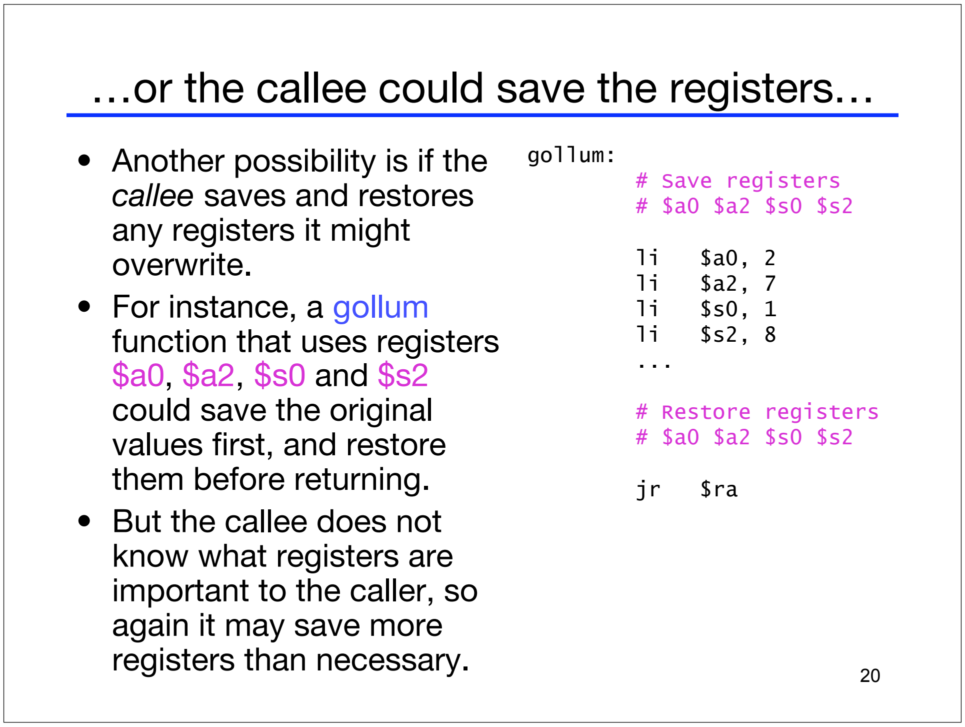
```
frodo: li   $a0, 3
       li   $a1, 1
       li   $s0, 4
       li   $s1, 1

       # Save registers
       # $a0, $a1, $s0, $s1

       jal  gollum

       # Restore registers
       # $a0, $a1, $s0, $s1

       add  $v0, $a0, $a1
       add  $v1, $s0, $s1
       jr   $ra
```



…or the callee could save the registers…

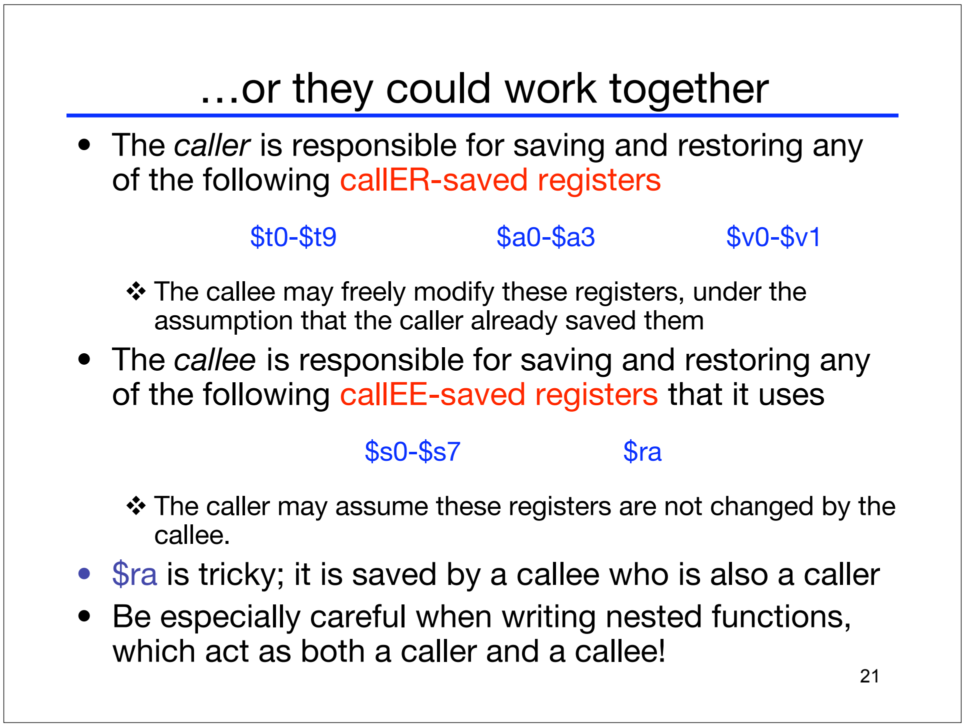
- Another possibility is if the *callee* saves and restores any registers it might overwrite.
- For instance, a gollum function that uses registers $a0, $a2, $s0 and $s2 could save the original values first, and restore them before returning.
- But the callee does not know what registers are important to the caller, so again it may save more registers than necessary.

```
gollum:
         # Save registers
         # $a0 $a2 $s0 $s2

         li   $a0, 2
         li   $a2, 7
         li   $s0, 1
         li   $s2, 8
         ...

         # Restore registers
         # $a0 $a2 $s0 $s2

         jr   $ra
```



…or they could work together

- The *caller* is responsible for saving and restoring any of the following callER-saved registers

  $t0-$t9 $a0-$a3 $v0-$v1

  - ❖ The callee may freely modify these registers, under the assumption that the caller already saved them
- The *callee* is responsible for saving and restoring any of the following callEE-saved registers that it uses

  $s0-$s7 $ra

  - ❖ The caller may assume these registers are not changed by the callee.
- $ra is tricky; it is saved by a callee who is also a caller
- Be especially careful when writing nested functions, which act as both a caller and a callee!