CS 305
Control Flow in MIPS

Control flow in high-level languages

- The instructions in a program usually execute one after another, but it's often necessary to alter the normal control flow.
- Conditional statements execute only if some test expression is true.

```
// Find the absolute value of *a0
v0 = *a0;
if (v0 < 0)
    v0 = -v0;           // This might not be executed
v1 = v0 + v0;
```

- Loops cause some statements to be executed many times.

```
// Sum the elements of a  five-element array a0
v0 = 0;
t0 = 0;
while (t0 < 5) {
    v0 = v0 + a0[t0];   // These statements will
    t0++;               // be executed five times
}
```

MIPS control instructions

- We have already seen some of MIPS branching and jump instructions

j	// for unconditional jumps
bne and beq	// for conditional branches
slt and slti	// set if less than (w/ and w/o an immediate)

What does this code do?

```
label:subi    $a0, $a0, 1
      bne     $a0, $zero, label
```

3

Recall: Pseudo-branch instructions in MIPS

- The MIPS processor only supports two conditional branch instructions, beq and bne (apart from blez and bgtz), but to simplify your life the assembler provides the following other branches:

```
blt     $t0, $t1, L1 // Branch if $t0 < $t1
ble     $t0, $t1, L2 // Branch if $t0 <= $t1
bgt     $t0, $t1, L3 // Branch if $t0 > $t1
bge     $t0, $t1, L4 // Branch if $t0 >= $t1
```

Translating an `if-then` statement

- We can use branch instructions to translate `if-then` statements into MIPS assembly code.

```
v0 = *a0;
if (v0 < 0)
    v0 = -v0;
v1 = v0 + v0;
```

```
move $v0 $a0
bge $v0, $0   Label
sub $v0, 0, $v0
Label: add $v1, $v0, $v0
```

- Sometimes it's easier to *invert* the original condition.
  - In this case, we changed "continue if v0 < 0" to "skip if v0 >= 0".
  - This saves a few instructions in the resulting assembly code.

Translating `if-then-else`

- If there is an else clause, it is the target of the conditional branch
  - And the then clause needs a jump over the else clause

```
// increase the magnitude of  v0 by one
if (v0 < 0)
    v0 --;

else
    v0 ++;
v1 = v0;
```

```
     bge  $v0, $0, E
     sub  $v0, $v0, 1
     j    L

E:   add  $v0, $v0, 1
L:   move $v1, $v0
```

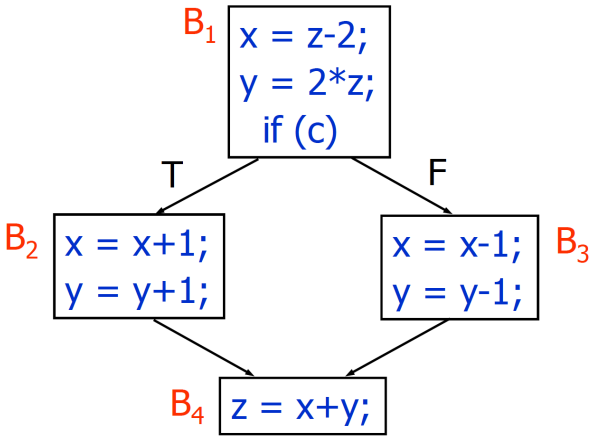
- Dealing with **else-if** code is similar, but the target of the first branch will be another if statement.
  - Drawing a control-flow graph can help you out.

CFGs

Program

```
x = z-2 ;
y = 2*z;
if (c) {
   x = x+1;
   y = y+1;
}
else {
   x = x-1;
   y = y-1;
}
z = x+y;
```

Control Flow Graph



Loops in MIPS ASM

```
for (i = 0; i < 4; i++) {
     // stuff
}
```

```
          add    $t0, $zero, $zero
 Loop:    // stuff
          addi   $t0, $t0, 1
          slti   $t1, $t0, 4
          bne    $t1, $zero, Loop
```

```
// these allocate storage
    int     i;
    int     N = 20;
    char    prompt[] = "Enter an integer:";
    int     A[MAX_SIZE];
    for (i=0; i<N; i++) {
         A[i] = MAX_SIZE; // MAX_SIZE=256
    }

    add     $t0, $gp, $zero      # address of i.
    lw      $t1, 4($gp)          # fetch N
    sll     $t1, $t1, 2          # N as byte offset
    add     $t1, $t1, $gp        # &A[N] - 28
    ori     $t2, $zero, 256      # MAX_SIZE
top:
    sltu    $t3, $t0, $t1        # have we reached the final
                                   address?
    beq     $t3, $zero, done     # yes, we're done
    sw      $t2, 28($t0)         # A[i] = 0
    addi    $t0, $t0, 4
                                 # el
    j       top                  # go to t
done:

# NOTE: We have not updated i in memory!
```

$t0 = gp
$t1 = gp + 80
$t2 = 256
Remember A starts at gp + 28

A is not being fetched here! We just store 256 in each A location starting with an offset of 28 from gp

Control-flow Example

- Let's write a program to **count how many bits are set in a 32-bit word.**

```
int count = 0;
for (int i = 0 ; i < 32 ; i ++) {
   int bit = input & 1;
   if (bit != 0) {
     count ++;
   }
   input = input >> 1;
}
```

```
.text
main:
        li      $a0, 0x1234     ## input = 0x1234
        li      $t0, 0          ## int count = 0;
        li      $t1, 0          ## for (int i = 0

main_loop:
        bge     $t1, 32, main_exit   ## exit loop if i >= 32

        andi    $t2, $a0, 1          ## bit = input & 1
        beq     $t2, $0, main_skip   ## skip if bit == 0

        addi    $t0, $t0, 1          ## count ++

main_skip:
        srl     $a0, $a0, 1          ## input = input >> 1
        add     $t1, $t1, 1          ## i ++

        j       main_loop

main_exit:
        jr      $ra
```

Notice the use of Pseudo instructions

Switch Statements in ASM

- Many high-level languages support multi-way branches, e.g.

```
switch (two_bits) {
    case 0: break;
    case 1: /* fall through */
    case 2:count ++; break;
    case 3: count += 2; break;
}
```

- We could just translate the code to if, thens, and elses:

```
if ((two_bits == 1) || (two_bits == 2)) {
        count ++;
} else if (two_bits == 3) {
        count += 2;
}
```

- This isn't very efficient if there are many, many cases.

Alternatively

- We can:

1. Create an array of jump targets
2. Load the entry indexed by the variable `two_bits`
3. Jump to that address using the jump register, or **jr**, instruction

```
switch (i) {
    case 0:
        A[0] = 0;
        break;
    case 1:
    case 2:
        A[1] = 1;
       break;
    default:
        A[0] = -1;
        break;
}
```

For this example, assume the compiler has generated a branch table and stored it after background in memory (i.e., starting at offset 1064 from $gp). The branch table is initialized to hold in successive locations the absolute addresses of the instructions at labels is0, is1, and is2.

```
    lw    $t0, 0($gp)          # fetch i
    bltz  $t0, def             # i<0 -> default
    slti  $t1, $t0, 3          # I < 3 ?
    beq   $t1, $zero, def      # no? -> default
    sll   $t0, $t0, 2          # turn i into a
                                     byte offset
    add   $t2, $t0, $gp
    lw    $t2, 1064($t2)       # fetch the
                                     branch table
                                     entry
    jr    $t2                  # go...

is0: sw    $zero, 28($gp)        # A[0] = 0
     j     done
is1: nop
is2: addi  $t0, $zero, 1         # = 1
     sw    $t0, 32($gp)          # A[1] = 1
     j     done
def: addi  $t0, $zero, -1        # = -1
     sw    $t0, 28($gp)          # A[0] = -1
     j     done
done:
```

```
int func( char* string ) {
    int count = 0;
    while( *string != '\0' ) {
        string++;
        count++;
    }
    return count;
}
```

For the MIPS assembly language version we will assume that count is stored in register t0 and that the address of the string is stored in register a0 (the first argument register).

```
   strlen:
      li $t0, 0           # initialize the count to zero
      loop:
         lbu $t1, 0($a0)  # load the next character into t1
         beqz $t1, exit   # check for the null character
         addi $a0, $a0, 1 # increment the string pointer
         addi $t0, $t0, 1 # increment the count
         j loop           # return to the top of the loop
   exit:
```

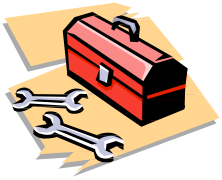
```
int max( int* array, int size )
{
    int maximum = array[0];
    for( int i=1;i<size;i++ )
        if( array[i] > maximum )
            maximum = array[i];
    return maximum;
}
```

Function Calls

Functions in MIPS

- 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.



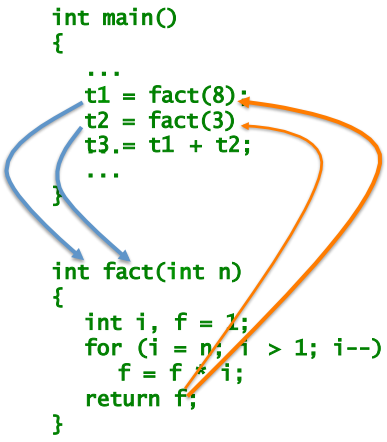
Control flow in C

- 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, in effect, called by the operating system when you run the program.



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 +8 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;
}
```

20

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.

21

Aside: A note about types

- Assembly language is untyped—there is no distinction between integers, characters, pointers or other kinds of values.
- 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?
  - CHAOS!

22

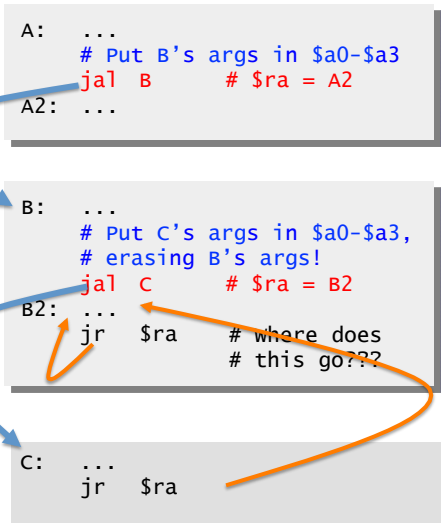
Issues with data flow so far

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

23

Nested functions

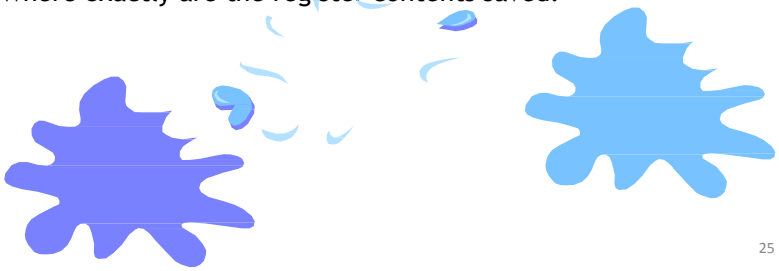
- A similar situation happens when you call a function that then calls another function.
- Let's say A calls B, which calls C.
  - The arguments for the call to C would be placed in $a0-$a3, thus *overwriting* the original arguments for B.
  - Similarly, jal C overwrites the return address that was saved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



24

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 and which ones do they save?
  - Where exactly are the register contents saved?



25

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?

26

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, main wants to preserve $a0, $a1, $s0 and $s1 from simpleFunction, but simpleFunction may not even use those registers.

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

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

      jal simpleFunction

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

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

27

…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.

```
simpleFunction:
        # 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
```

28

…or they could work together

- MIPS uses conventions again to split the register spilling chores.
- The *caller* is responsible for saving and restoring any of the following caller-saved registers that it cares about.

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

- In other words, the callee may freely modify these registers, under the assumption that the caller already saved them if necessary.
- The *callee* is responsible for saving and restoring any of the following callee-saved registers that it uses.

$s0-$s7

- Thus the caller may assume these registers are not changed by the callee.
- Be especially careful when writing nested functions, which act as both a caller and a callee!

29

How about $ra?

- Recall that a jal instruction saves something into $ra and therefore overwrites whatever was there earlier.

Register spilling example

- This convention ensures that the caller and callee together save all of the important registers—frodo only needs to save registers $a0 and $a1, while gollum only has to save registers $s0 and $s2.

```
Func1:  li    $a0, 3
        li    $a1, 1
        li    $s0, 4
        li    $s1, 1

        # Save registers
        # $a0, $a1, $ra

        jal   Func2

        # Restore registers
        # $a0, $a1, $ra

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

```
Func2:
        # Save registers
        # $s0 and $s2

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

        # Restore registers
        # $s0 and $s2

        jr    $ra
```

31

How to fix factorial

- In the factorial example, main (the caller) should save two registers.
  - $t1 must be saved before the second call to fact.
  - $ra will be implicitly overwritten by the jal instructions.
- But **fact** (the callee) does not need to save anything. It only writes to registers $t0, $t1 and $v0, which should have been saved by the caller.

```
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;
}
```

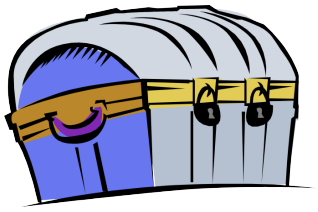
32

So far….

- Control flow changes in MIPS assembly

Where are the registers saved?

- Now we know who is responsible for saving which registers, but we still need to discuss **where** those registers are saved.
- It would be nice if each function call had its own private memory area.
  - This would prevent other function calls from overwriting our saved registers—otherwise using memory is no better than using registers.
  - We could use this private memory for other purposes too, like storing local variables.



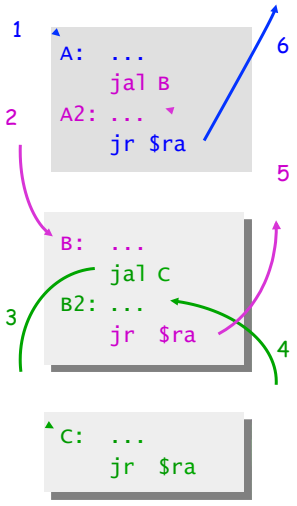
34

Function calls and stacks

- Notice function calls and returns occur in **a stack-like order**: the most recently called function is the first one to return.

  1. Someone calls A
  2. A calls B
  3. B calls C
  4. C returns to B
  5. B returns to A
  6. A returns

- Here, for example, C must return to B *before* B can return to A.



35

Stacks and function calls

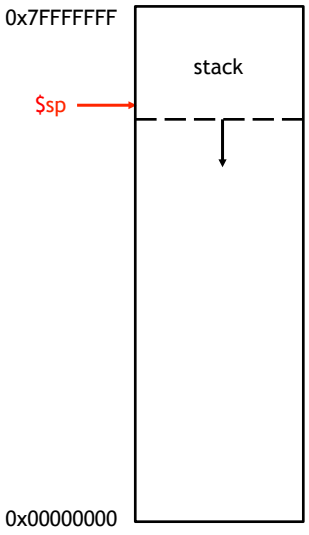
- It's natural to use a stack for function call storage. A block of stack space, called a stack frame, can be allocated for each function call.
  - When a function is called, it creates a new frame onto the stack, which will be used for local storage.
  - Before the function returns, it must pop its stack frame, to restore the stack to its original state.

- The stack frame can be used for several purposes.
  - Caller- and callee-save registers can be put in the stack.
  - The stack frame can also hold local variables, or extra arguments and return values.

36

The MIPS stack

- In MIPS machines, part of main memory is reserved for a stack.
  - The stack grows downward in terms of memory addresses.
  - The address of the top element of the stack is stored (by convention) in the "stack pointer" register, $sp.
- MIPS does not provide "push" and "pop" instructions. Instead, they must be done explicitly by the programmer.



37

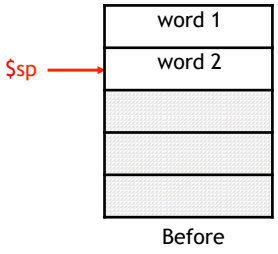
Pushing elements

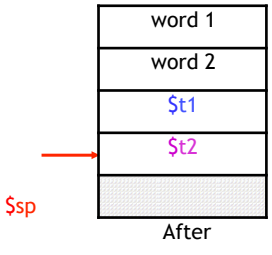
- To push elements onto the stack:
  - Move the stack pointer $sp down to make room for the new data.
  - Store the elements into the stack.
- For example, to push registers $t1 and $t2 onto the stack:

```
sub  $sp,  $sp,  8
sw   $t1,  4($sp)
sw   $t2,  0($sp)
```

- An equivalent sequence is:

```
sw   $t1,  -4($sp)
sw   $t2,  -8($sp)
sub  $sp,  $sp,  8
```





38

Accessing and popping elements

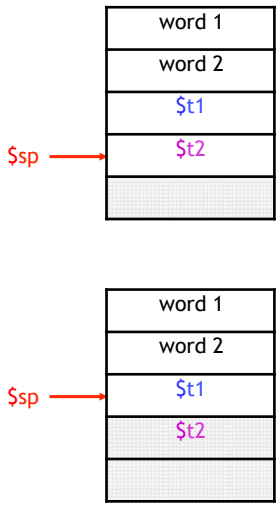
- You can access any element in the stack (not just the top one) if you know where it is relative to $sp.
- For example, to retrieve the value of $t1:

```
lw   $s0, 4($sp)
```

- You can pop, or "erase," elements simply by adjusting the stack pointer upwards.
- To pop the value of $t2, yielding the stack shown at the bottom:

```
addi $sp, $sp, 4
```

- Note that the popped data is still present in memory, but data past the stack pointer is considered invalid.



39

An example

```
int main() {
   int s0 = 5;
   int t0 = 10;
   int s1;
   s1 = function(s0);
   s1 = s1 + t0;
   s1 = s1*4;
   return 0;
}
```

```
int function(int a) {
   int s0, t0 = 15;
   s0 = a + t0;
   s0 = 3*s0;
   return s0;
}
```

Summary

- Implementing function calls in MIPS.
  - —We call functions using jal, passing arguments in registers $a0-$a3.
  - —Functions place results in $v0-$v1 and return using jr $ra.
- Managing resources is an important part of function calls.
  - —To keep important data from being overwritten, registers are saved according to conventions for caller-save and callee-save registers.
  - —Each function call uses stack memory for saving registers, storing local variables and passing extra arguments and return values.
- Assembly programmers must follow many conventions. Nothing prevents a rogue program from overwriting registers or stack memory used by some other function.

41

Implementing a Recursive Function

- Suppose we want to implement this in MIPS:

```
int fact (int n)
{
    if (n < 1)
       return (1);
    else
       return (n * fact (n - 1));
}
```

- It is a **recursive** function – a function that calls itself.
- It will keep on calling itself, with different parameters, until a terminating condition is met.

Recursion

What happens if we call fact(4)?

- First time call fact, compare 4 with 1, no less than 1, **call** fact again – fact(3).
- Second time call fact, compare 3 with 1, no less than 1, **call** fact agai – fact(2).
- Third time call fact, compare 2 with 1, no less than 1, **call** fact again – fact(1).
- Fourth time call fact, compare 1 with 1, no less than 1, **call** fact again – fact(0).
- Fifth time call fact, compare 0 with 1, less than 1, **return 1.**
- Return to the time when fact(0) was called (during the call of fact(1)). Multiply 1 with 1, **return 1.**
- Return to the time when fact(1) was called (during the call of fact(2)). Multiply 2 with 1, **return 2.**
- Return to the time when fact(2) was called (during the call of fact(3)). Multiply 3 with 2, **return 6.**
- Return to the time when fact(3) was called (during the call of fact(4)). Multiply 4 with 6, **return 24**.

```
int fact (int n)
{
    if (n < 1)
       return (1);
    else
       return (n * fact (n - 1));
}
```

The Recursive Function

- In MIPS, we say calling a function as going to the function. So we go to the function over and over again, until the terminating condition is met.
- Here, the function is called "fact," so we will have a line of code inside the fact function:

  `jal fact`

```
fact: jal fact
```

The Recursive Function

- The parameter should be passed in `$a0`. In the C function, every time we call fact, we call with n-1. So, in the MIPS function, before we do "`jal fact`", we should have "`addi $a0, $a0,-1.`"

```
fact: addi $a0, $a0, -1
      jal fact
```

The Recursive Function

- After calling fact, we multiply the return result with n, so, need to add multiplications.

```
fact: addi $a0, $a0, -1
      jal fact
      mul $v0, $v0, $a0
```

The Recursive Function

- After multiplying, we return.

```
fact: addi $a0, $a0, -1
      jal fact
      mul $v0, $v0, $a0
      jr $ra
```

The Recursive Function

- So, one if-else branch is done. The other branch is to compare `$a0` with 1, and should call `fact` again if less than 1 and otherwise return 1.

```
fact:  slti $t0, $a0, 1
       beq $t0, $zero, L1
       ori $v0, $0, 1
       jr $ra
L1:    addi $a0, $a0, -1
       jal fact
       mul $v0, $v0, $a0
       jr $ra
```

Any problems?

The Recursive Function

- The problem is that the function will call itself, as we have expected, but it will not return correctly!
- We need to save `$ra`, because we made another function call inside the function. We should always do so.
- Is this enough?

```
fact: addi $sp, $sp, -4
   sw $ra, 0($sp)
   slti $t0, $a0, 1
   beq $t0, $zero, L1
   ori $v0, $0, 1
   lw $ra, 0($sp)
   addi $sp, $sp, 4
   jr $ra
L1:   addi $a0, $a0, -1
   jal fact
   mul $v0, $v0, $a0
   lw $ra, 0($sp)
   addi $sp, $sp, 4
   jr $ra
```

The Recursive Function

- So now we can return to the main function, but the return result is 0, why?
- A call to `fact` modifies `$a0`. But when we return from a call, we multiply it with `$a0`!
- So, should also save `$a0`!
- Restore it before using it again.

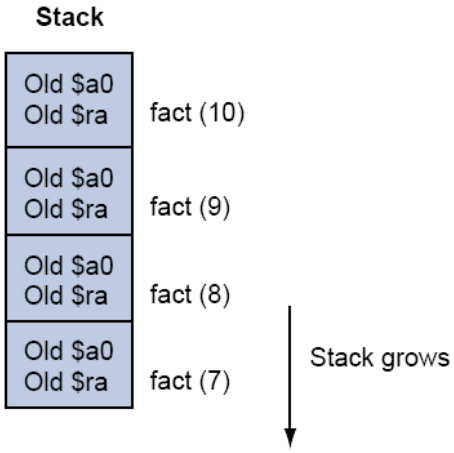
```
fact: addi $sp, $sp, -8
   sw $ra, 4($sp)
   sw $a0, 0($sp)
   slti $t0, $a0, 1
   beq $t0, $zero, L1
   ori $v0, $0, 1
   addi $sp, $sp, 8
   jr $ra
L1:   addi $a0, $a0, -1
   jal fact
   lw $ra, 4($sp)
   lw $a0, 0($sp)
   mul $v0, $v0, $a0
   addi $sp, $sp, 8
   jr $ra
```

```
        .text
        .globl main
Main:   li $a0, 4
        jal fact

done:   li $v0,10
        syscall

fact:   addi $sp, $sp, -8
        sw $ra, 4($sp)
        sw $a0, 0($sp)
        slti $t0, $a0, 1
        beq $t0, $zero, L1
        ori $v0, $0, 1
        addi $sp, $sp, 8
        jr $ra
L1:     addi $a0, $a0, -1
        jal fact
        lw $ra, 4($sp)
        lw $a0, 0($sp)
        mul $v0, $v0, $a0
        addi $sp, $sp, 8
        jr $ra
```

The Stack During Recursion



52

Two other MIPS pointers

- `$fp`: When you call a C function, the function may declare an array of size 100 like int A[100]. It is on the stack. You would want to access it, but the stack pointer may keep changing, so you need a fixed reference. `$fp` is the "frame pointer," which should always point to the first word that is used by this function.
- `$gp`: the "global pointer." A reference to access the static data.

Recursion in MIPS (single function call)

- Suggestions for simply implementing recursive function calls in MIPS
  1. Handle the base case first
     - Before you allocate a stack frame if possible
  2. Allocate stack frame
  3. Save return address
  4. Recursive Body:
     a) Save any registers needed after the call
     b) Compute arguments
     c) Call function
     d) Restore any registers needed after the call
     e) Consume return value (if any)
  5. Deallocate stack frame and return.

54

Recursion in MIPS (multiple function calls – caller save)

- Suggestions for simply implementing recursive function calls in MIPS
  1. Handle the base case first
     - Before you allocate a stack frame if possible
  2. Allocate stack frame
  3. Save return address
  4. For each function call: *(suggestion: use **$s** registers if >1 call)*
     a) Save any registers needed after the call
     b) Compute arguments
     c) Call function
     d) Restore any registers needed after the call
     e) Consume return value (if any)
  5. Deallocate stack frame and return.

55

Recursion in MIPS (multiple function calls – callee save)

- Suggestions for simply implementing recursive function calls in MIPS
  1. Handle the base case first
     - Before you allocate a stack frame if possible
  2. Allocate stack frame
  3. Save return address
  4. Save enough **$s** registers to hold your local variables
  5. Copy your local variables to **$s** registers
  6. For each function call:
     a) ~~Save any registers needed after the call~~
     b) Compute arguments
     c) Call function
     d) ~~Restore any registers needed after the call~~
     e) Consume return value (if any)
  7. Restore **$s** registers
  8. Deallocate stack frame and return.

56