CS250
Computer Architecture

Fall 2012

Part 5: More MIPS Instructions and Function Calls

Outline

- We'll go into more detail about the ISA.
  - Pseudo-instructions
  - Using branches for conditionals

Pseudo-instructions

- MIPS assemblers support pseudo-instructions that give the illusion of a more expressive instruction set, but are actually translated into one or more simpler, “real” instructions.
- Examples: the li and move pseudo-instructions:

```
li    $a0, 2000        # Load immediate 2000 into $a0
move  $a1, $t0         # Copy $t0 into $a1
```

- They are probably clearer than their corresponding MIPS instructions:

```
addi  $a0, $0, 2000    # Initialize $a0 to 2000
add   $a1, $t0, $0     # Copy $t0 into $a1
```

- We’ll see lots more pseudo-instructions this semester.
  - Unless otherwise stated, you can always use pseudo-instructions in your assignments and on exams.

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
}
```

Control-flow graphs

- It can be useful to draw control-flow graphs when writing loops and conditionals in assembly:

```
// Find the absolute value of *a0
v0 = *a0;
if (v0 < 0)
    v0 = -v0;
v1 = v0 + v0;
```

```
// Sum the elements of a0
v0 = 0;
t0 = 0;
while (t0 < 5) {
    v0 = v0 + a0[t0];
    t0++;
}
```

MIPS control instructions

- In section, we introduced some of MIPS's control-flow instructions

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

- And how to implement loops

- Today, we'll talk about
  – MIPS's pseudo branches
  – if/else

What does this code do?

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

Pseudo-branches

- The MIPS processor only supports two branch instructions, beq and bne, 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
```

- There are also immediate versions of these branches, where the second source is a constant instead of a register.

- Later this semester we'll see how supporting just beq and bne simplifies the processor design.

Implementing pseudo-branches

- Most pseudo-branches are implemented using slt. For example, a branch-if-less-than instruction `blt $a0, $a1, Label` is translated into the following.

```
slt  $at, $a0, $a1     // $at = 1 if $a0 < $a1
bne  $at, $0, Label    // Branch if $at != 0
```

- This supports immediate branches, which are also pseudo-instructions. For example, `blti $a0, 5, Label` is translated into two instructions.

```
slti $at, $a0, 5       // $at = 1if $a0 < 5
bne  $at, $0, Label    // Branch if $a0 < 5
```

- All of the pseudo-branches need a register to save the result of slt, even though it's not needed afterwards.
  - MIPS assemblers use register $1, or $at, for temporary storage.
  - You should be careful in using $at in your own programs, as it may be overwritten by assembler-generated code.

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;
```

- 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.

Control-flow Example

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

Translating an if-then-else statements

- 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

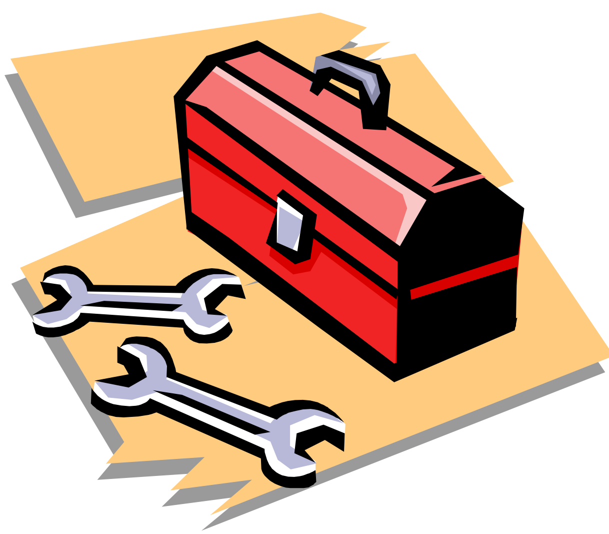
```
// increase the magnitude of v0 by one
if (v0 < 0)
    v0 --;

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 the control-flow graph can help you out.

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, in effect, 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
```

- Let's now add the jal and jr instructions that are necessary for our factorial example.

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.
- 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.

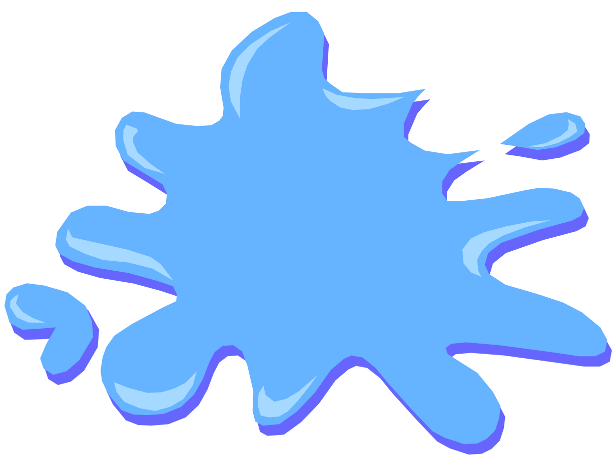
```
A:   ...
     # Put B's args in $a0-$a3
     jal  B       # $ra = A2
A2:  ...
```

```
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

- 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. (Remember that $ra is "used" by jal.)

  $s0-$s7 $ra

  Thus 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!

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.

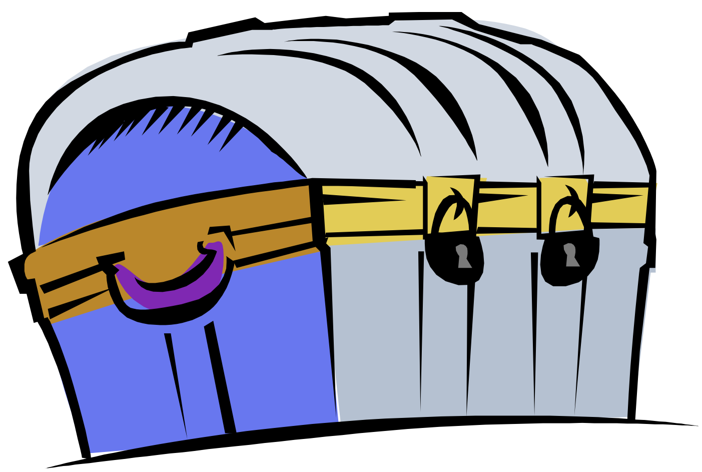
```
frodo:  li    $a0, 3                  gollum:
        li    $a1, 1                            # Save registers
        li    $s0, 4                            # $s0 and $s2
        li    $s1, 1
                                                li    $a0, 2
        # Save registers                        li    $a2, 7
        # $a0, $a1, $ra                         li    $s0, 1
                                                li    $s2, 8
        jal   gollum                            ...

        # Restore registers                     # Restore registers
        # $a0, $a1, $ra                         # $s0 and $s2

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

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.

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.

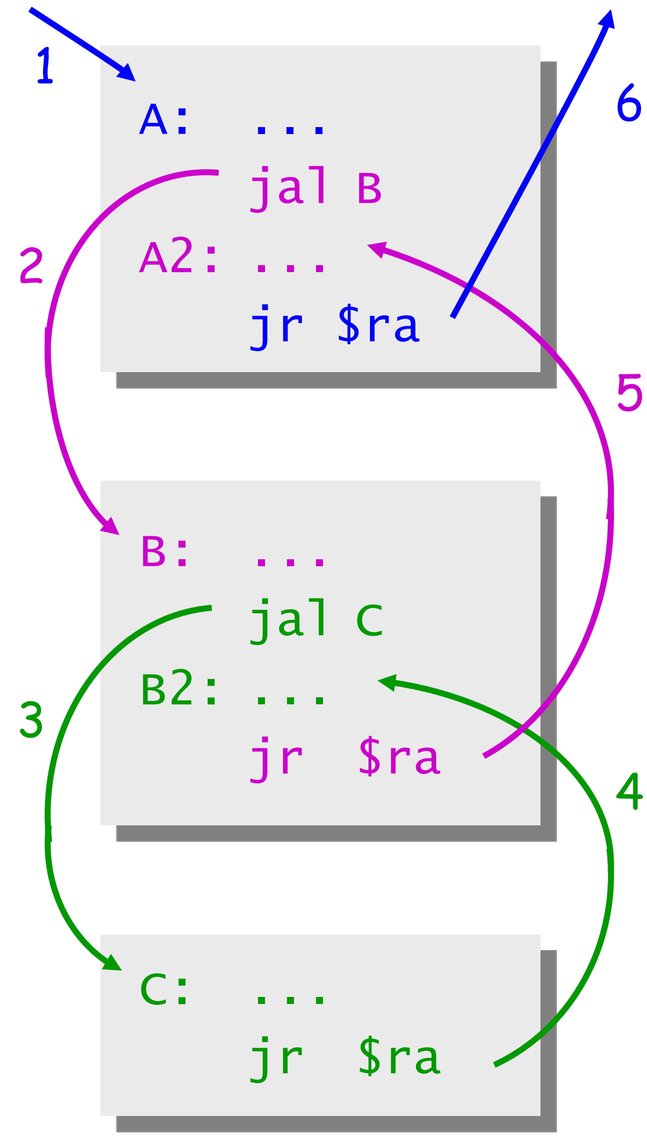


Function calls and stacks

- 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.

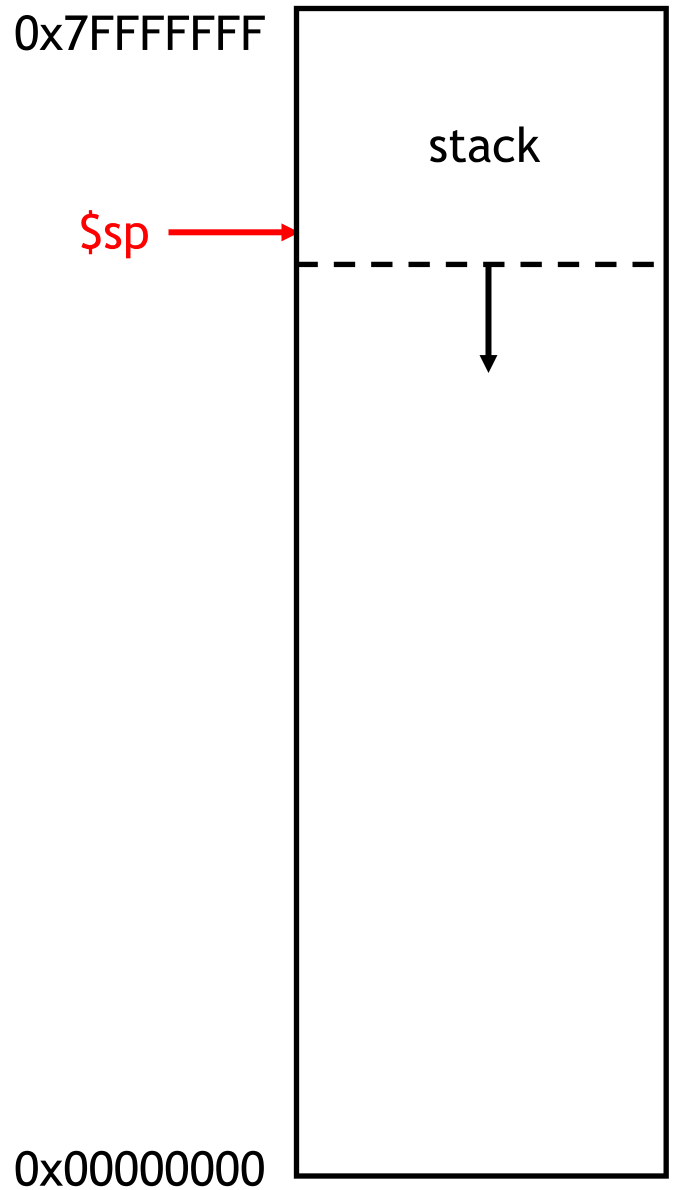


Stacks and function calls

- 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.

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.



Pushing elements

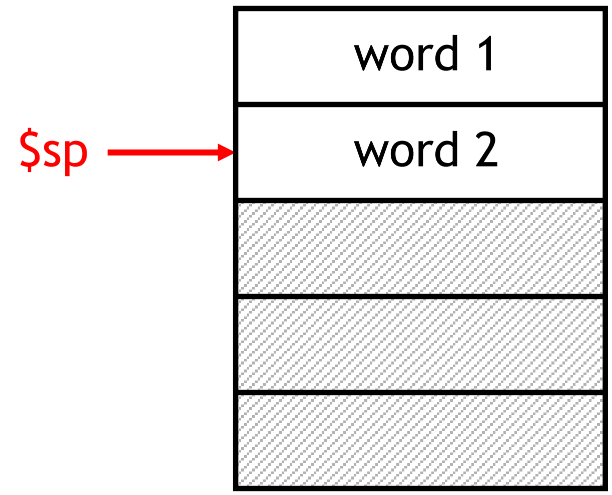
- To push elements onto the stack:
  - Move the stack pointer $sp down to make room for the new data.
  - Store the elements into the stack.
- For example, to push registers $t1 and $t2 onto the stack:

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

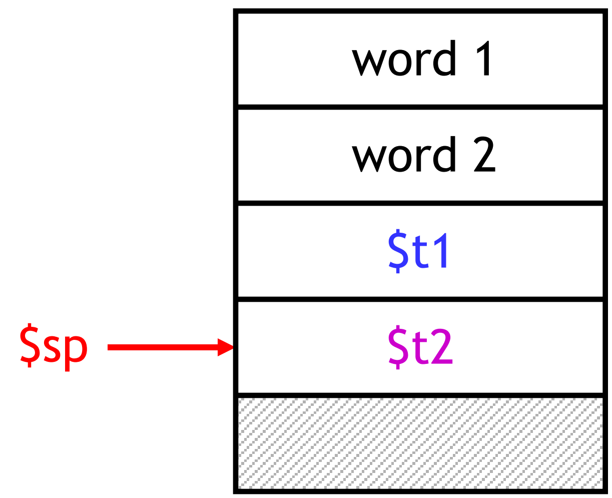
- An equivalent sequence is:

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

- Before and after diagrams of the stack are shown on the right.



Before

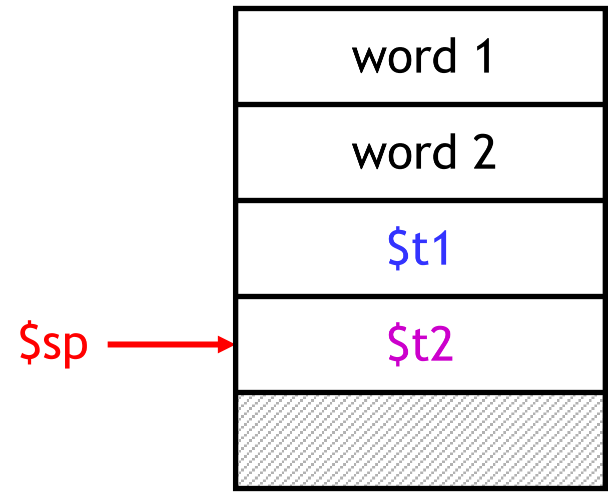


After

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:

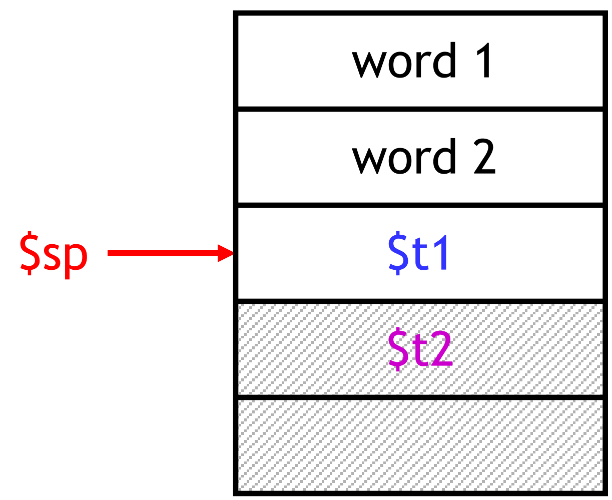
```
lw   $s0, 4($sp)
```



- 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:

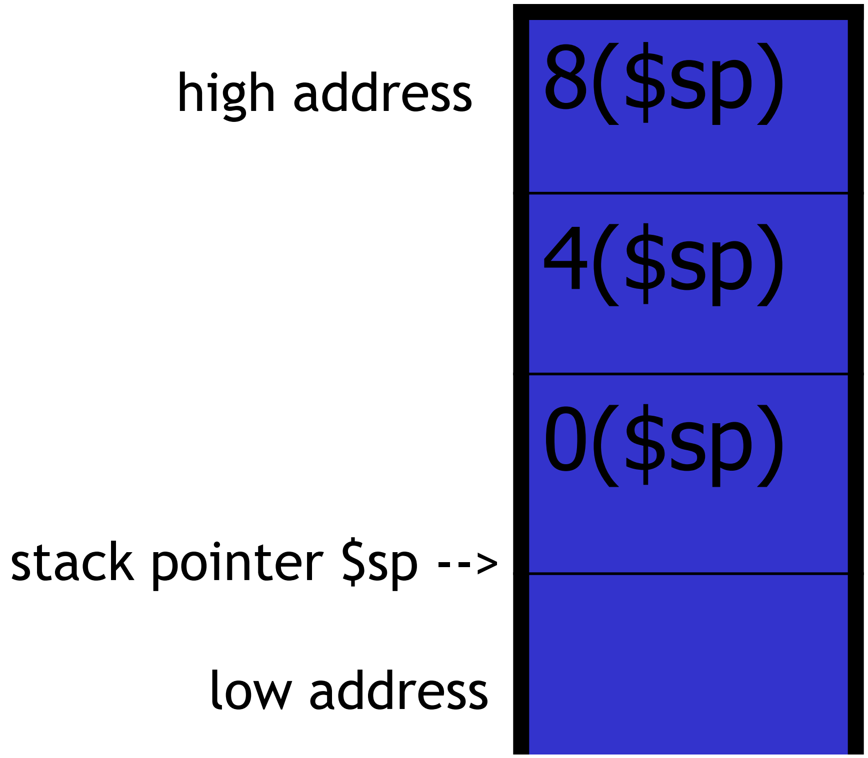
```
addi $sp, $sp, 4
```

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



Stack

The stack can be used for
- storing return addresses
- storing register values
- parameter passing
- ...



$sp = $sp - 12

Factorial

- Compute n!
- Recall that
  - 0! = 1
  - 1! = 1
  - n! = n(n-1)!
- Store on the stack
  - $s0 = n, the parameter
  - $ra, the return address

```
factorial:
      bgt $a0,$0,gen     # if $a0>0 goto generic case
      li $v0, 1          # base case, 0! = 1
      jr $ra             # return

gen:  subi $sp,$sp,8     # create a stack frame
      sw $s0,0($sp)      # store register $s0’s value
      sw $ra,4($sp)      # store return address
      move $s0,$a0       # save argument
      subi $a0,$a0,1     # factorial(n-1)
      jal factorial      # v0 = (n-1)!
      mul $v0,$s0,$v0    # n*(n-1)!
      lw $s0,0($sp)      # restore $s0=n
      lw $ra,4($sp)      # restore $ra
      addi $sp,$sp,8     # destruct the stack frame
      jr $ra             # return
```

Fibonacci

fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)

0, 1, 1, 2, 3, 5, 8, 13, 21,…

Fibonacci

```
li $a0, 10          # call fib(10)
jal fib             #
move $s0, $v0       # $s0 = fib(10)
```

fib is another recursive procedure with one argument $a0
need to store argument $a0, temporary register $s0 for intermediate results, and return address $ra

```
fib:    subi $sp,$sp,12          # save registers on stack
        sw $a0, 0($sp)           # save $a0 = n
        sw $s0, 4($sp)           # save $s0
        sw $ra, 8($sp)           # save return address $ra
        bgt $a0,1, gen           # if n>1 then goto generic case
        move $v0,$a0             # output = input if n=0 or n=1
        j rest                   # goto restore registers
gen:    subi $a0,$a0,1           # param = n-1
        jal fib                  # compute fib(n-1)
        move $s0,$v0             # save fib(n-1)
        sub $a0,$a0,1            # set param to n-2
        jal fib                  # and make recursive call
        add $v0, $v0, $s0        # $v0 = fib(n-2)+fib(n-1)
rest:   lw  $a0, 0($sp)          # restore registers from stack
        lw  $s0, 4($sp)          #
        lw  $ra, 8($sp)          #
        addi $sp, $sp, 12        # decrease the stack size
        jr $ra
```

Summary

- Today we focused on 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.