EECE 2322: Fundamentals of Digital Design and Computer Organization

Lecture 8_2: MIPS ISA

Xiaolin Xu
Department of ECE
Northeastern University

Function Control Flow of MIPS

- MIPS uses the jump-and-link instruction jal to call functions.
  - 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**

Input Arguments & Return Value

MIPS assembly code

```
# $s0 = y

main:
  ...
  addi $a0, $0, 2     # argument 0 = 2
  addi $a1, $0, 3     # argument 1 = 3
  addi $a2, $0, 4     # argument 2 = 4
  addi $a3, $0, 5     # argument 3 = 5
  jal  diffofsums     # call Function
  add  $s0, $v0, $0   # y = returned value
  ...

# $s0 = result
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  jr  $ra             # return to caller
```

Input Arguments & Return Value

MIPS assembly code

```
# $s0 = result
diffofsums:
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result = (f + g) - (h + i)
  add $v0, $s0, $0   # put return value in $v0
  jr  $ra            # return to caller
```

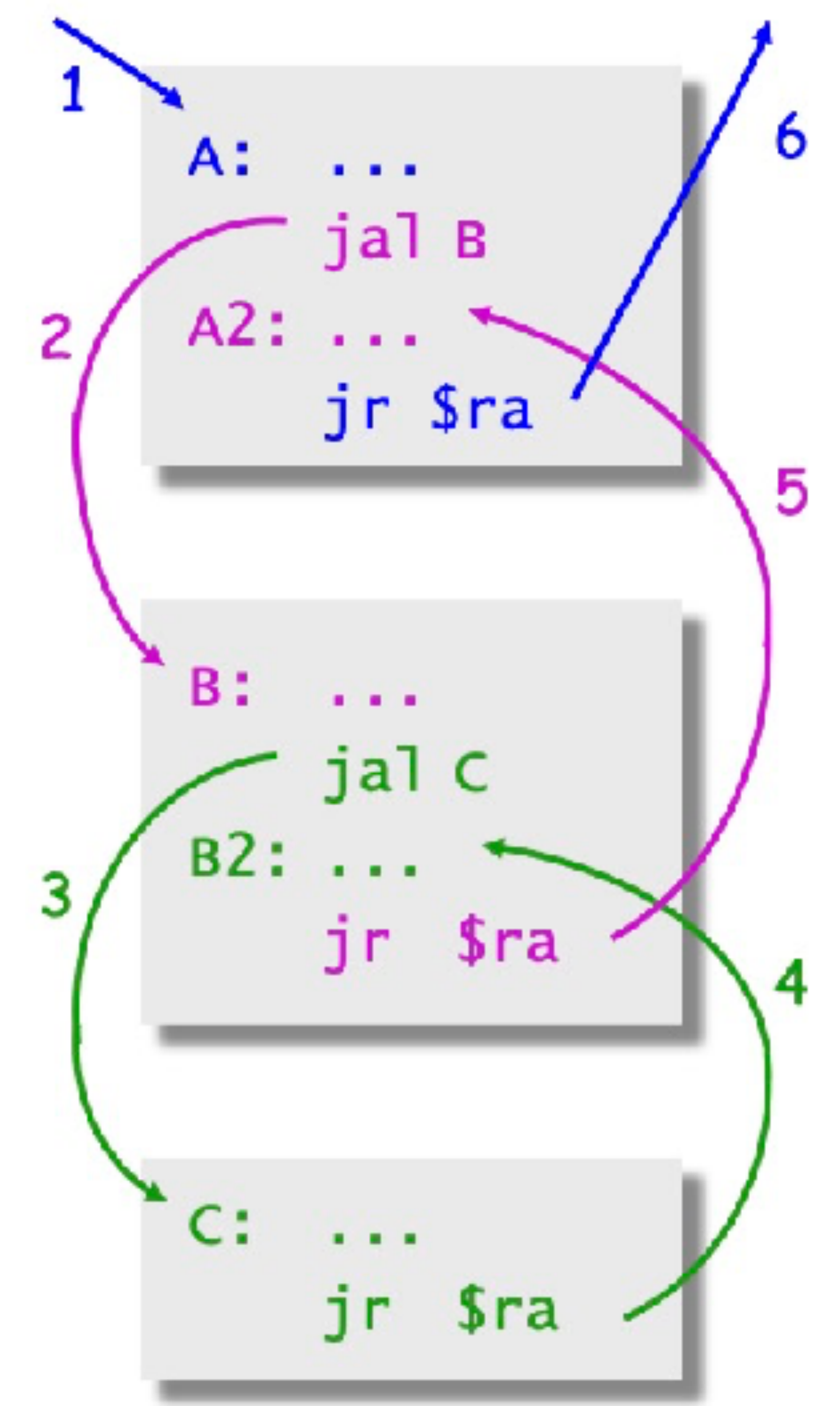
- `diffofsums` overwrote 3 registers: `$t0`, `$t1`, `$s0`
- `diffofsums` can use *stack* to temporarily store registers

Practical Function Call

- Function calls and returns: the most recently called function is the first one to return

Practical Function Call

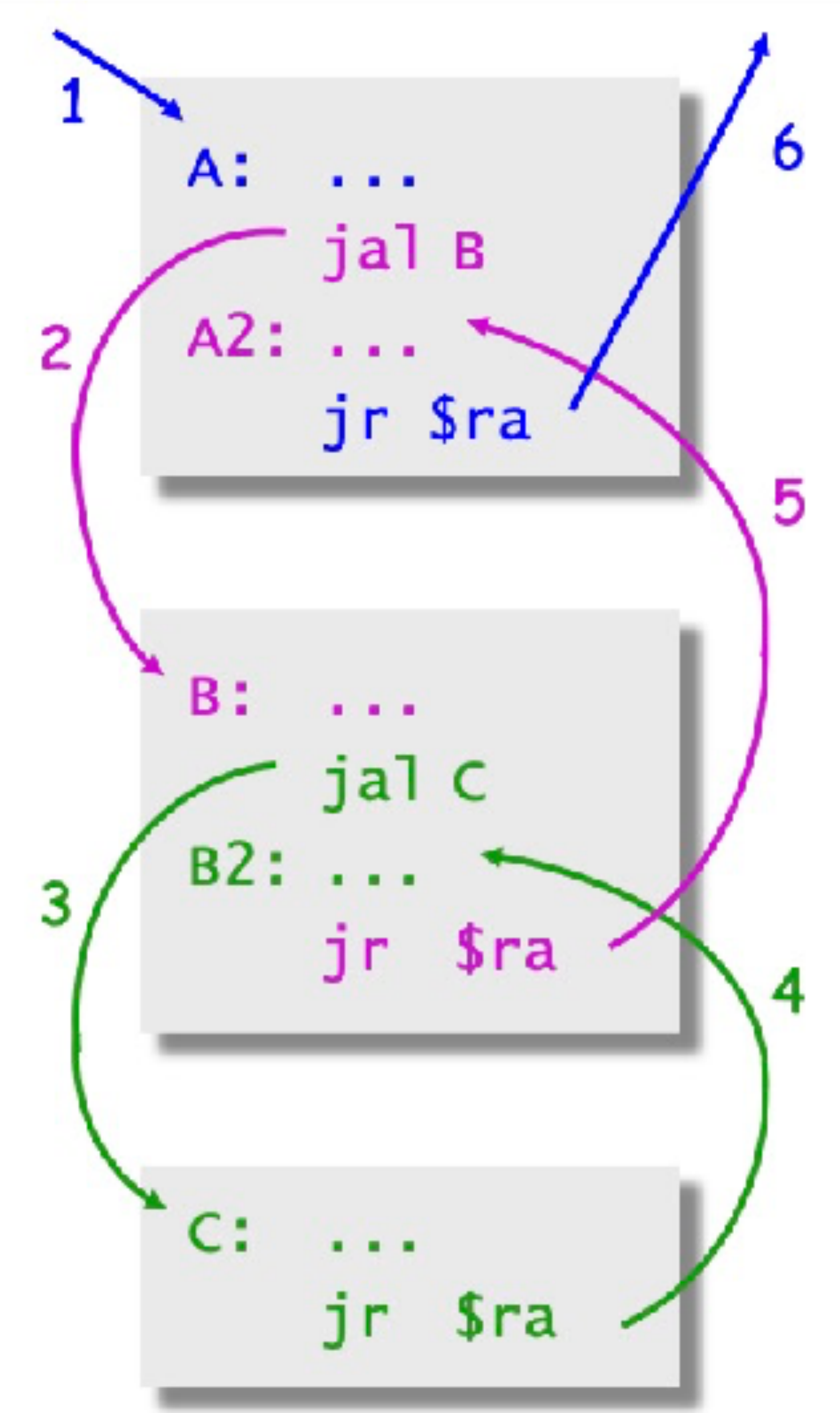
- Function calls and returns: the most recently called function is the first one to return



Practical Function Call

- Function calls and returns: the most recently called function is the first one to return

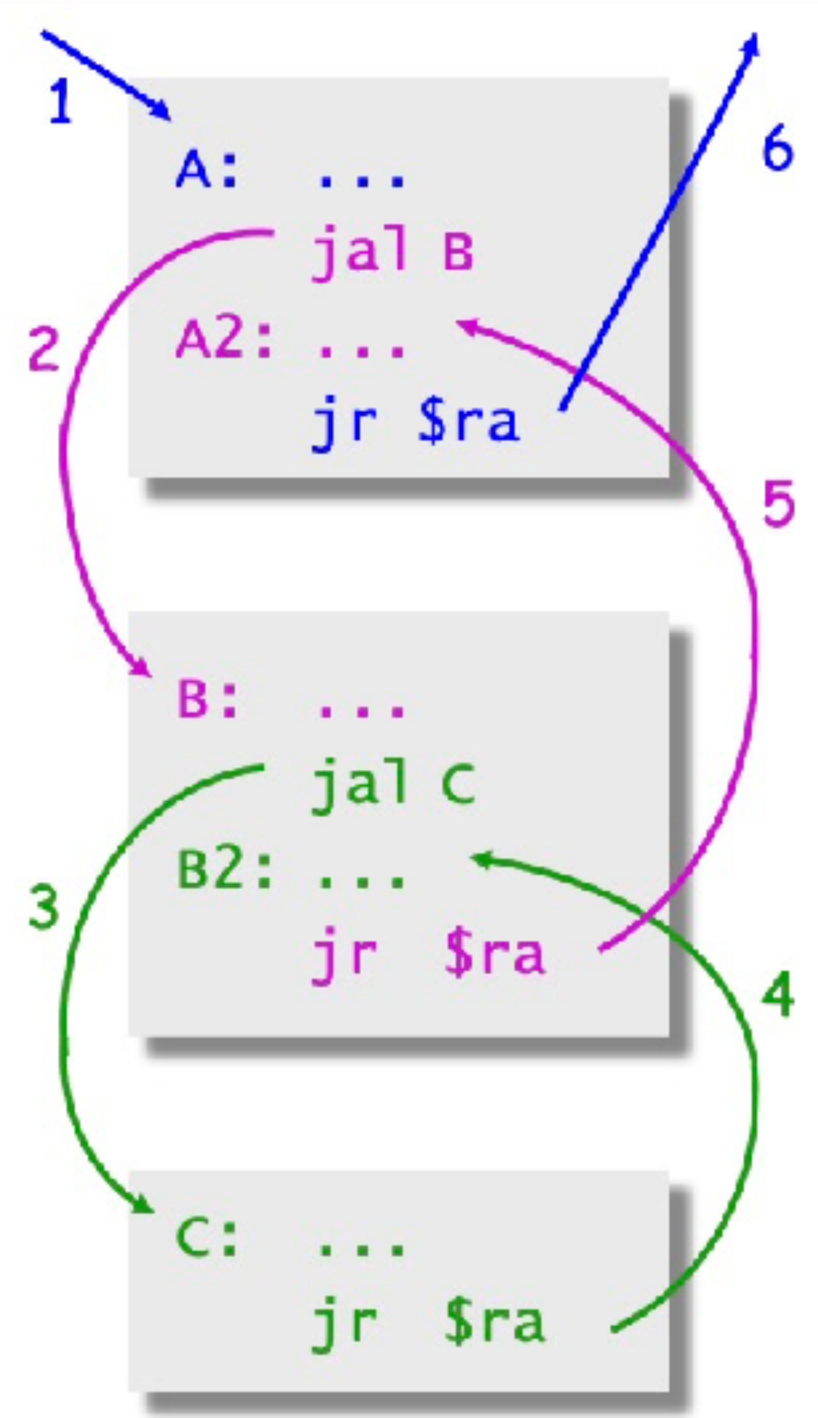
1. Someone calls A
2. A calls B
3. B calls C
4. C returns to B
5. B returns to A
6. A returns



Practical Function Call

- Function calls and returns: 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



- A stack-like order: C must return to B before B can return to A

Example Function: Factorial

- **BLEZ** -- Branch on less than or equal to zero
  - Branches if the register is less than or equal to zero

```
int fact(int n) {

  int i, f = 1;
  for (i = n; i > 0; i--)
    f = f * i;
    return f;
}
```

```
fact:
        li $t0, 1
        move $t1,$a0          # set i to n
loop:
        blez $t1,exit         # exit if done
        mul $t0,$t0,$t1       # build factorial
        addi $t1, $t1,-1      # i--
        j loop
exit:
        move $v0,$t0
        jr $ra
```

Calling the Factorial Function

- **BLEZ** -- Branch on less than or equal to zero
  - Branches if the register is less than or equal to zero

```
int fact(int n) {

  int i, f = 1;
  for (i = n; i > 0; i--)
    f = f * i;
    return f;
}
```

```
fact:
        li $t0, 1
        move $t1,$a0        # set i to n
loop:
        blez $t1,exit       # exit if done
        mul $t0,$t0,$t1     # build factorial
        addi $t1, $t1,-1    # i--
        j loop
exit:
        move $v0,$t0
        jr $ra
```

Calling the Factorial Function

- A function might be called several times, while itself uses a number of registers
- How to avoid conflict?
  - Limited number of registers—> shared by all functions!

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

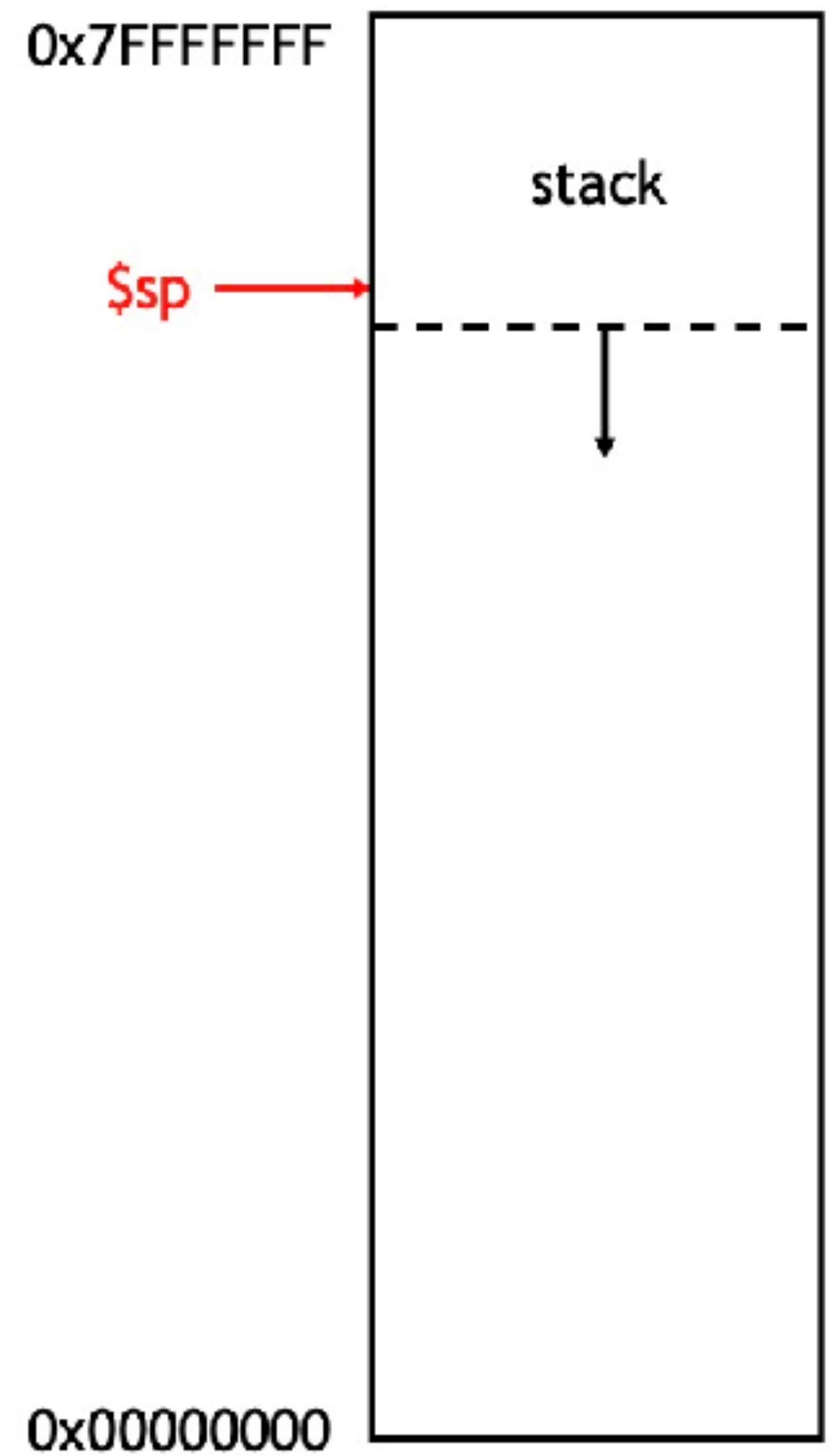

int fact(int n)
{
   int i, f = 1;
   for (i = n; i > 1; i--)
      f = f * i;
   return f;
}
```

The Stack

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- ***Expands***: uses more memory when more space needed
- ***Contracts***: uses less memory when the space is no longer needed

The Stack in MIPS

- Part of main memory is reserved for a stack
- MIPS does not provide “push” and “pop” instructions. Instead, they must be done explicitly by the programmer.



The Stack in MIPS

- Grows down (from higher to lower memory addresses)
- Stack pointer: `$sp` points to top of the stack

Address	Data	
7FFFFFFC	12345678	← $sp
7FFFFFF8		
7FFFFFF4		
7FFFFFF0		
⋮	⋮	

Address	Data	
7FFFFFFC	12345678	
7FFFFFF8	AABBCCDD	
7FFFFFF4	11223344	← $sp
7FFFFFF0		
⋮	⋮	

How Functions use the Stack

- Called functions must have no unintended side effects
  - Recall that `diffofsums` overwrites 3 registers: `$t0`, `$t1`, `$s0`
  - What if these registers are already used before calling `diffofsums`

```
# MIPS assembly
# $s0 = result
diffofsums:
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  jr  $ra             # return to caller
```

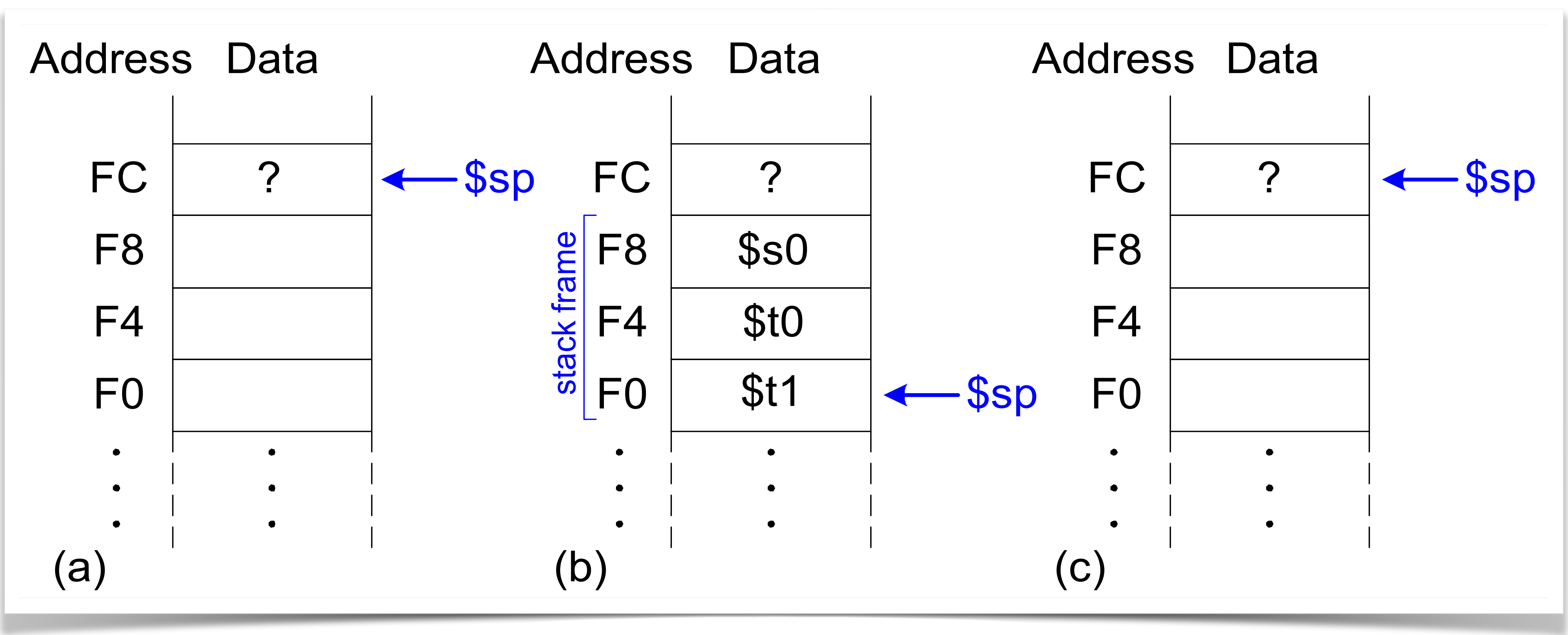
Solution

- 5 Steps
  - 1. Makes space on the stack to store the values of one or more registers.
  - 2. Stores the values of the registers on the stack.
  - 3. Executes the function using the registers.
  - 4. Restores the original values of the registers from the stack.
  - 5. Deallocates space on the stack.

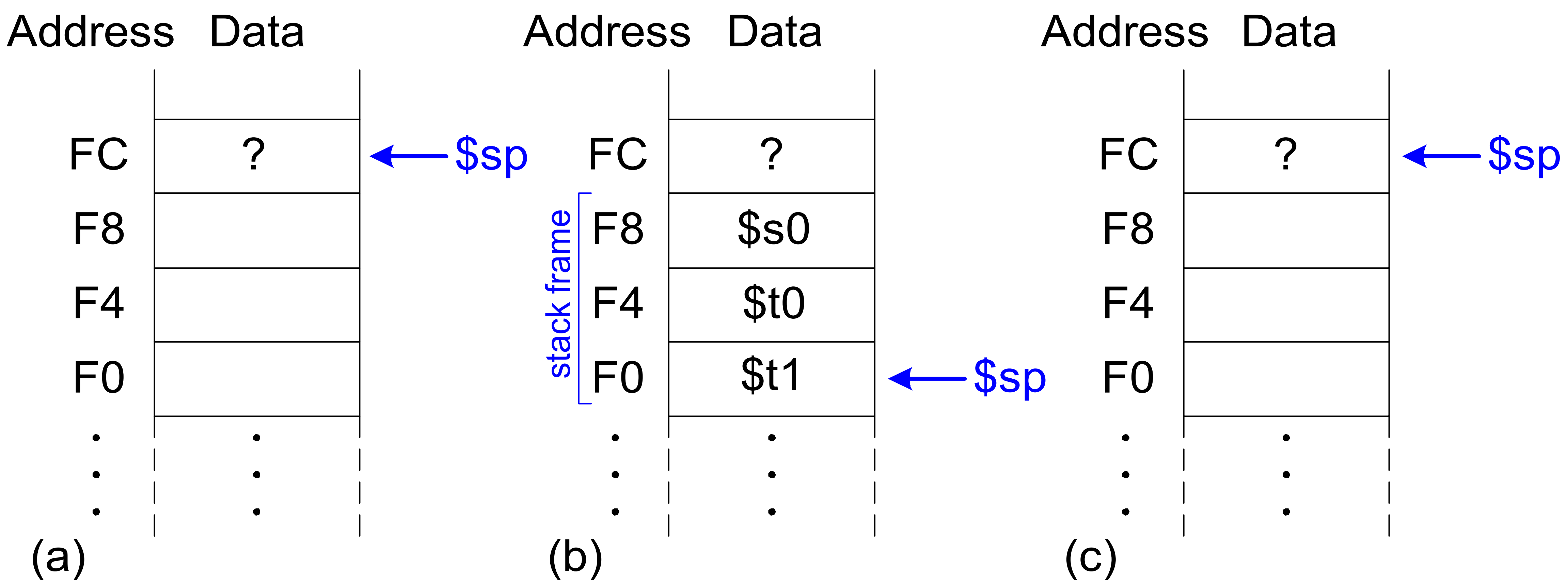
Storing Register Values on the Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -12  # make space on stack
                      # to store 3 registers
  sw   $s0, 8($sp)    # save $s0 on stack
  sw   $t0, 4($sp)    # save $t0 on stack
  sw   $t1, 0($sp)    # save $t1 on stack
  add  $t0, $a0, $a1  # $t0 = f + g
```

……………

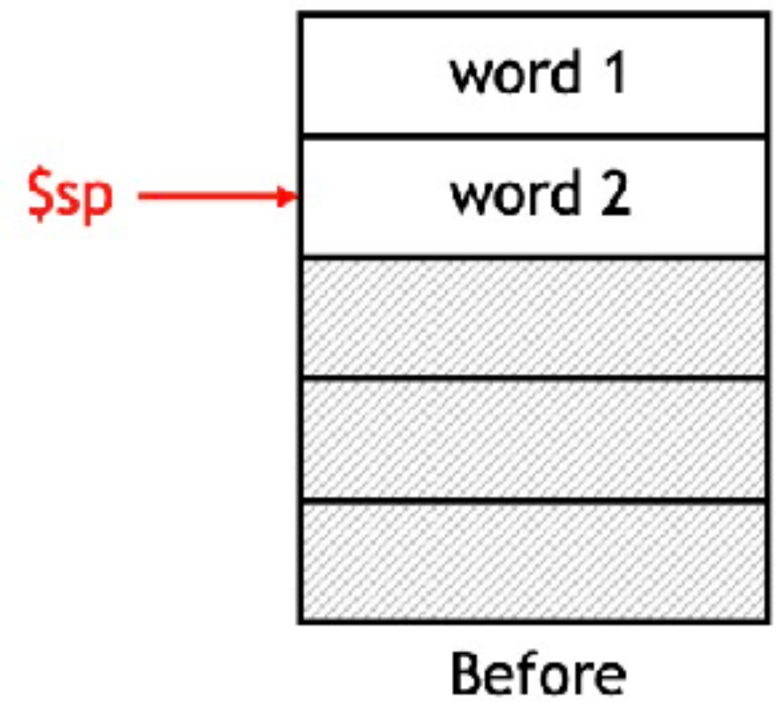


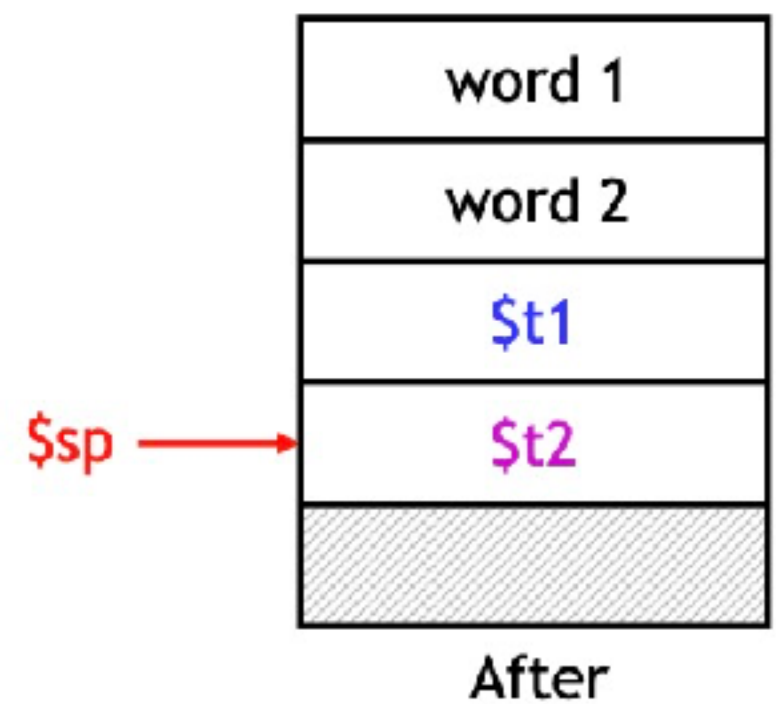
The stack during diffofsums Call



"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:
-
```
addi $sp, $sp, -8
sw   $t1, 4($sp)
sw   $t2, 0($sp)
```



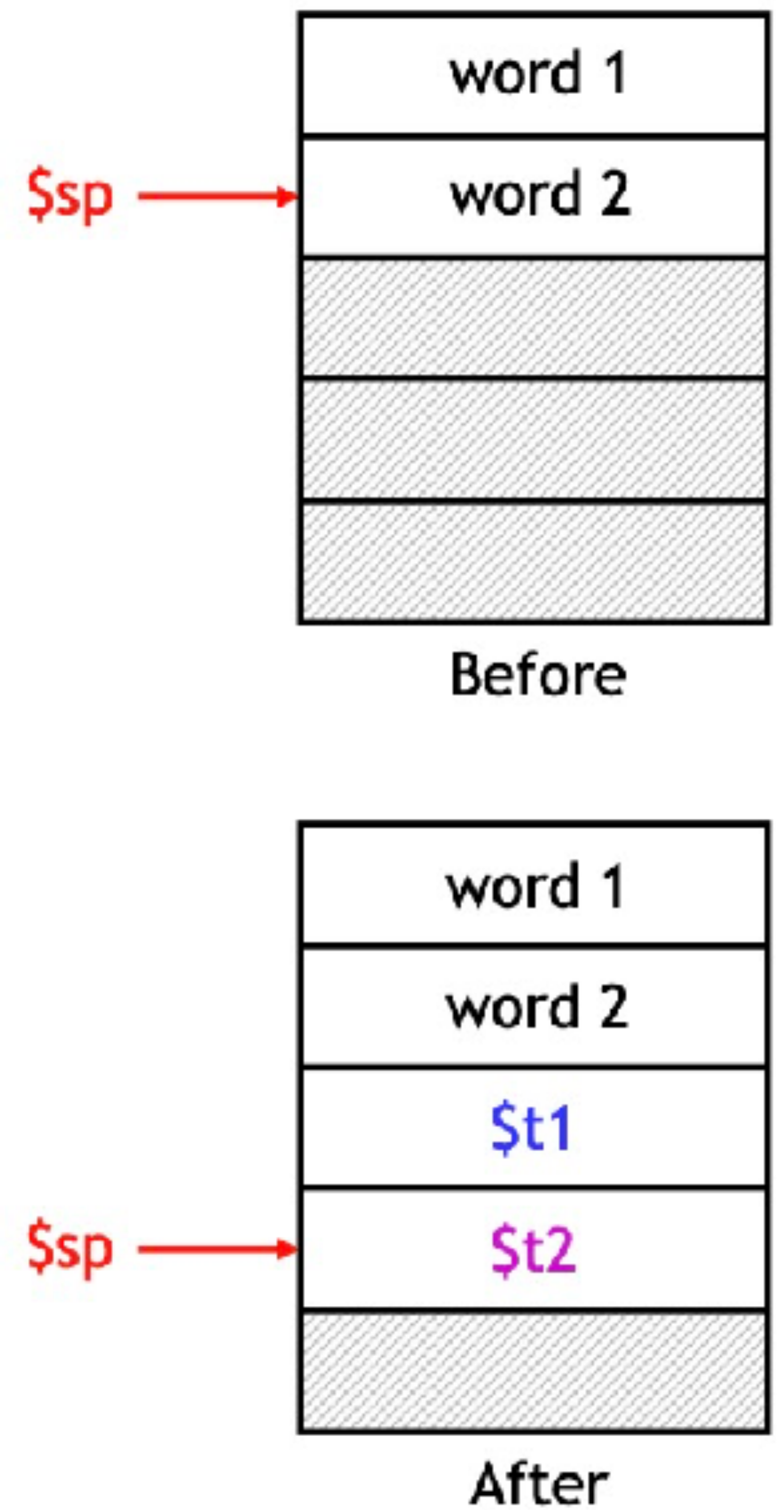


"Pushing" Elements

- 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.
- Equivalent stack operations:

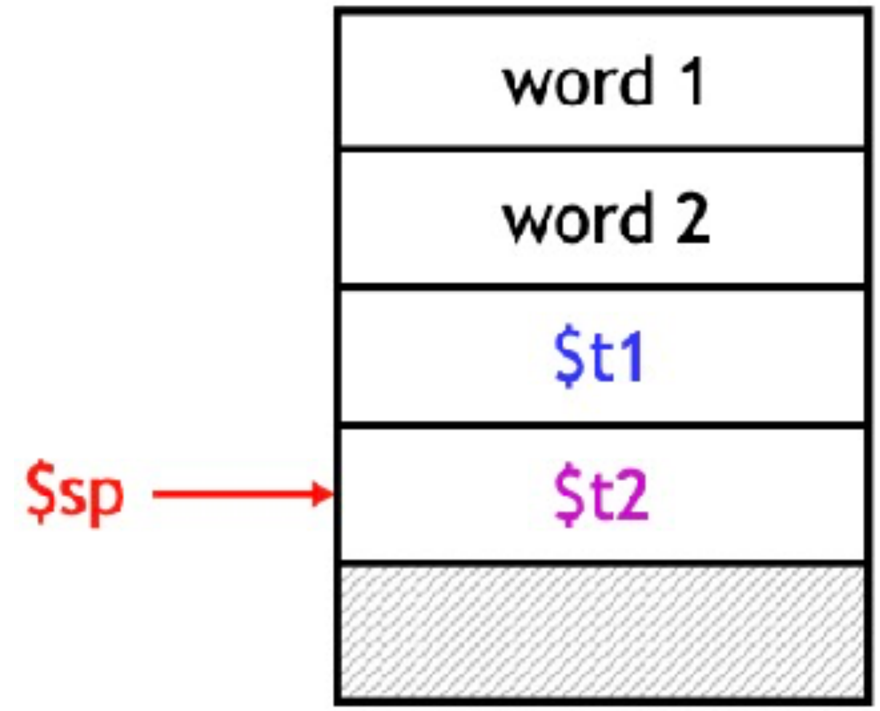
```
sw   $t1, -4($sp)
sw   $t2, -8($sp)
```

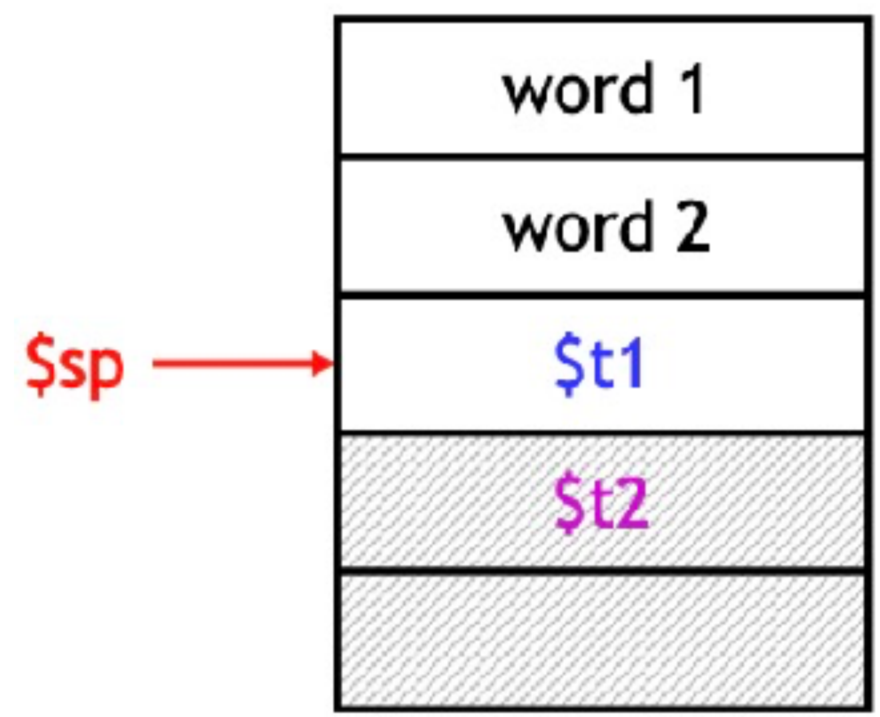
- ```
  addi $sp, $sp, -8
  ```


  ```

Accessing Elements in Stack

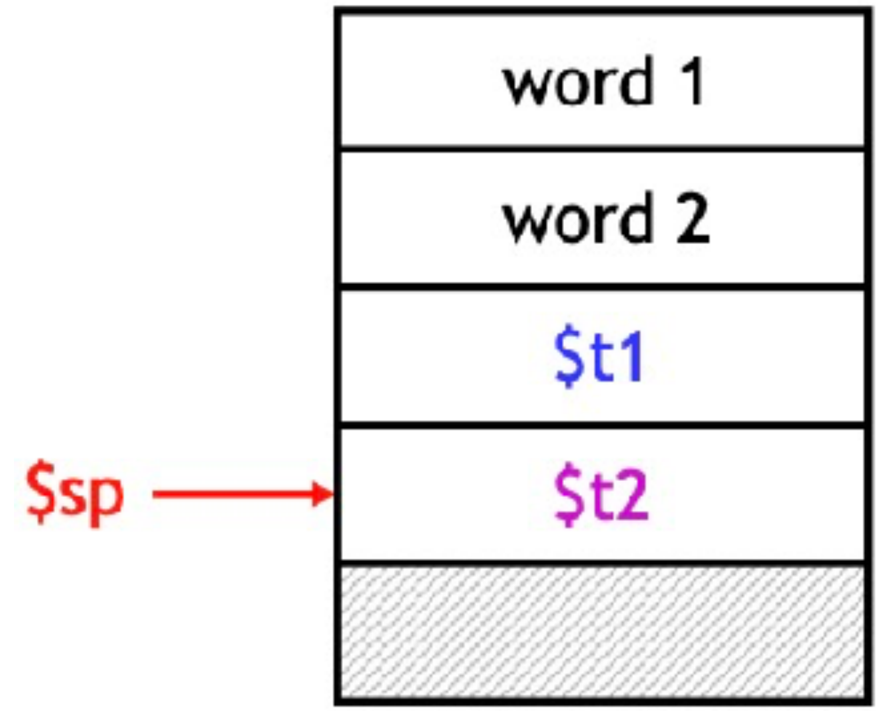
- A user can access to any stack element, Not just the top one —> pointed by $sp!
- But, s/he should know the location, i.e., relative to $sp!
- For example, to retrieve the value of $t1:
  - **lw $s0, 4($sp)**

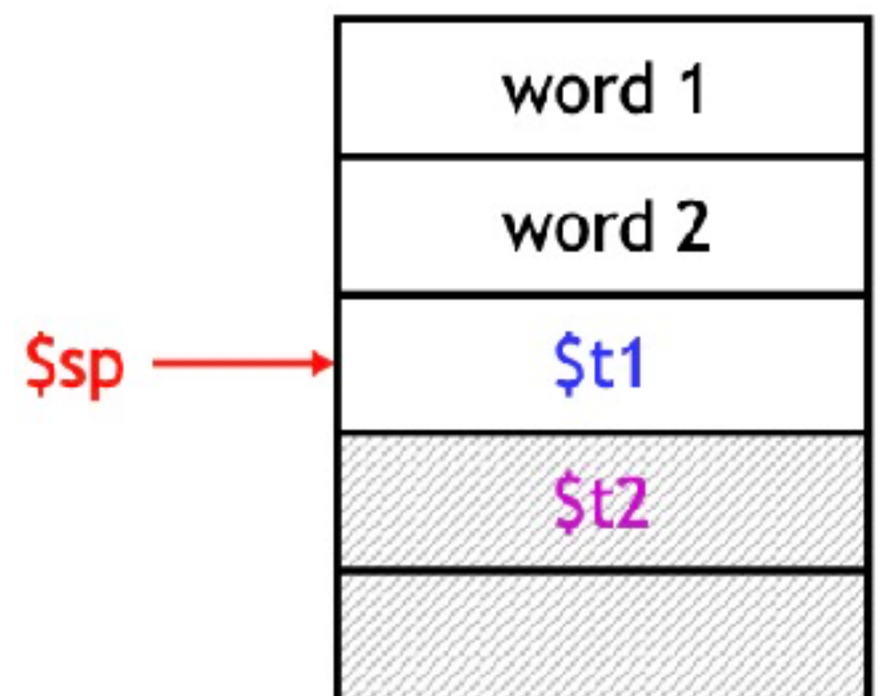




Accessing Elements in Stack

- To pop the value of $t2, yielding the stack shown at the bottom:
  - **addi $sp, $sp, 4**
  - **Note that $t2 is still valid in the memory, if you want to erase it, reset the stack point!**





Storing Register Values on the Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -12   # make space on stack
                       # to store 3 registers
  sw   $s0, 8($sp)     # save $s0 on stack
  sw   $t0, 4($sp)     # save $t0 on stack
  sw   $t1, 0($sp)     # save $t1 on stack
  add  $t0, $a0, $a1   # $t0 = f + g
  add  $t1, $a2, $a3   # $t1 = h + i
  sub  $s0, $t0, $t1   # result = (f + g) - (h + i)
  add  $v0, $s0, $0    # put return value in $v0
  lw   $t1, 0($sp)     # restore $t1 from stack
  lw   $t0, 4($sp)     # restore $t0 from stack
  lw   $s0, 8($sp)     # restore $s0 from stack
  addi $sp, $sp, 12    # deallocate stack space
  jr   $ra             # return to caller
```

Storing Register Values on the Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -12   # make space on stack
                       # to store 3 registers
  sw   $s0, 8($sp)     # save $s0 on stack
  sw   $t0, 4($sp)     # save $t0 on stack
  sw   $t1, 0($sp)     # save $t1 on stack
  add  $t0, $a0, $a1   # $t0 = f + g
  add  $t1, $a2, $a3   # $t1 = h + i
  sub  $s0, $t0, $t1   # result = (f + g) - (h + i)
  add  $v0, $s0, $0    # put return value in $v0
  lw   $t1, 0($sp)     # restore $t1 from stack
  lw   $t0, 4($sp)     # restore $t0 from stack
  lw   $s0, 8($sp)     # restore $s0 from stack
  addi $sp, $sp, 12    # deallocate stack space
  jr   $ra             # return to caller
```

- **Doable, but too complicated and low-efficiency!**

Who Should Save the Registers in Stack?

- Caller
  - Possibility: the caller to save any important registers that it needs before making a function call, and to restore them after

Who Should Save the Registers in Stack?

- Caller
  - Possibility: the caller to save any important registers that it needs before making a function call, and to restore them after
- Problem?

Who Should Save the Registers in Stack?

- Caller
  - Possibility: the caller to save any important registers that it needs before making a function call, and to restore them after
- Problem?
  - The caller does not know what registers are actually written by the function, so it may save more registers than necessary.

Who Should Save the Registers in Stack?

- Callee
  - Callee saves and restores any registers it might overwrite

Who Should Save the Registers in Stack?

- Callee
  - Callee saves and restores any registers it might overwrite
- Problem?

Who Should Save the Registers in Stack?

- Callee
  - Callee saves and restores any registers it might overwrite
- Problem?
  - Callee does not know what registers are important to the caller, so again it may save more registers than necessary

Quick Summary of the register Saving Issue

- 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?

How about “Working Together”?

- Assumption
  - Some of the registers are not changed by the caller, others not changed by the callee
  - Making it much easier for the compiler, as well as for the programmer

How about "Working Together"?

- Asking Caller and callee to each be responsible for a specific set of registers
- *Caller* is responsible for saving and restoring any of the following caller-saved registers that it cares about.
  - **\$t0-\$t9 \$a0-\$a3 \$v0-\$v1**
  - **Callee may freely modify these registers, under the assumption that the caller already saved them if necessary.**
- *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**

Storing Saved Registers on the Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -4  # make space on stack to
            # store one register
  sw   $s0, 0($sp)   # save $s0 on stack
                     # no need to save $t0 or $t1
  add $t0, $a0, $a1  # $t0 = f + g
  add $t1, $a2, $a3  # $t1 = h + i
  sub $s0, $t0, $t1  # result = (f + g) - (h + i)
  add $v0, $s0, $0   # put return value in $v0
  lw   $s0, 0($sp)   # restore $s0 from stack
  addi $sp, $sp, 4   # deallocate stack space
  jr   $ra           # return to caller
```

Storing Saved Registers on the Stack

```
# $s0 = result
diffofsums:
  addi $sp, $sp, -4   # make space on stack to
                # store one register
  sw   $s0, 0($sp)    # save $s0 on stack
                      # no need to save $t0 or $t1
  add $t0, $a0, $a1   # $t0 = f + g
  add $t1, $a2, $a3   # $t1 = h + i
  sub $s0, $t0, $t1   # result = (f + g) - (h + i)
  add $v0, $s0, $0    # put return value in $v0
  lw   $s0, 0($sp)    # restore $s0 from stack
  addi $sp, $sp, 4    # deallocate stack space
  jr   $ra            # return to caller
```

Revisiting the Factorial Function

```c
int fact(int n) {

  int i, f = 1;
  for (i = n; i > 0; i--)
    f = f * i;
    return f;
}
```

```
fact:
        li $t0, 1
        move $t1,$a0        # set i to n
loop:
        blez $t1,exit       # exit if done
        mul $t0,$t0,$t1     # build factorial
        addi $t1, $t1,-1    # i--
        j loop
exit:
        move $v0,$t0
        jr $ra
```

Revisiting the Factorial Function

```c
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;
}
```

```
li $t0, 1
move $t1,$a0          # set i to n

blez $t1,exit         # exit if done
mul $t0,$t0,$t1       # build factorial
addi $t1, $t1,-1      # i--
j loop

move $v0,$t0
jr $ra
```

Revisiting the Factorial Function

```
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;
}
```

- $t1 will be saved before the 2nd call
- $ra is overwritten by jal

```
li $t0, 1
move $t1,$a0          # set i to n

blez $t1,exit         # exit if done
mul $t0,$t0,$t1       # build factorial
addi $t1, $t1,-1      # i--
j loop

move $v0,$t0
jr $ra
```

Summary

- Any values in the preserved registers MUST be saved-and-restored, while a function can change the non-preserved registers freely!
  - —> making it more efficient
  - The **caller saves any non-preserved registers ($t0–$t9 and $a0–$a3)** that are needed after the call.
  - The **callee saves any of the pre- served registers ($s0–$s7 and $ra)** that it intends to modify
- **However!**
- **What if a function uses non-preserved registers?**

Summary

- It is the **caller's responsibility** to save-and-restore any values in the **non-preserved registers**
- It is the **callee's responsibility** to save-and-restore any values in the **preserved registers**
- **Like priority over which type of registers)**

Preserved *Callee-Saved*	Nonpreserved *Caller-Saved*
`$s0-$s7`	`$t0-$t9`
`$ra`	`$a0-$a3`
`$sp`	`$v0-$v1`
stack above `$sp`	stack below `$sp`